SOCKET MIGRATION FOR OPENMOSIX

by

ETHAN BOWKER

B.S., Kansas State University, 2003

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2008

Approved by:

Major Professor

Dwight Day

Abstract

Process migration is a technique in clustering and distributed computing by which parallel applications can be dynamically moved between nodes in a cluster in response to differing phases of execution, which is of growing usefulness in the field of distributed computing. A drawback to many recent implementations of process migration is that sockets for interprocess communication do not migrate with the process requiring communication to be rerouted through the process' starting, or home, node, resulting in reduced communications performance when the process is migrated away from its home node.

This thesis focuses on the implemention a solution to this problem at the kernel level for the OpenMosix process migration system with efficient socket handoff and cluster-wide unique addressing by reimplemting TCP on top of the existing network code in the Linux kernel.

Although falling short of the intial goal of fully transparent operation, this thesis presents a working implementation of migratable sockets for the OpenMosix process migration system that demonstrates working socket migration and improved performance over non-migrating sockets in OpenMosix.

Table of Contents

Table of Contents iii

List of Figures v

List of Tables vi

Acknowledgements vii

1 Introduction 1
- 1.1 MOSIX and OpenMosix . . . 1
- 1.2 Origin of Project . . . 2

2 TCP/IP and Sockets 4
- 2.1 TCP/IP Protocol Suite . . . 4
- 2.2 Berkeley Sockets . . . 6
- 2.3 TCP . . . 8

3 Design 12
- 3.1 Design Goals . . . 12
- 3.2 Related Work . . . 13
- 3.3 Practical Considerations . . . 15
- 3.4 MTCP . . . 16
- 3.5 Metasockets . . . 17

4 Implementation 19
- 4.1 Organization . . . 19
- 4.2 Sockets . . . 21
- 4.3 Packets . . . 26
  - 4.3.1 Structures . . . 26
  - 4.3.2 Lower-Level Routines . . . 28
  - 4.3.3 Routing . . . 29
- 4.4 Interface to Existing IP Stack . . . 29
- 4.5 Socket Migration . . . 30
  - 4.5.1 Migration Daemon . . . 31
  - 4.5.2 Metasockets . . . 31

4.5.3 Miscellaneous Migration Handling 33

5 Testing and Performance **35**
5.1 Userspace Unit Tests 35
5.2 Kernel Tests 36
5.3 Performance 37

6 Conclusion and Future Work **42**
6.1 Conclusion 42
6.2 Future Work 43

A Kernel API **47**

B User API **55**
B.1 ms_calls.h 56
B.2 ms_calls.c 57

C Client/Server Example **60**
C.1 test_common.h 60
C.2 test_common.c 61
C.3 client2.c 63
C.4 server2.c 69

List of Figures

1.1 DESPOT system architecture 3

2.1 TCP/IP protocol layers 4
2.2 UDP header layout 5
2.3 TCP header 9
2.4 TCP handshake 10
2.5 TCP states 11

3.1 MTCP protocol layering 16
3.2 MTCP header layout 17

4.1 The connection fields of the socket structure 22
4.2 The status fields of the socket structure, along with corresponding value definitions 23
4.3 The sequence variables for the socket structure 23
4.4 Sychronization and threading variables for the socket structure 24
4.5 Socket list/list entry fields in the socket structure 25
4.6 Definition of the `ms_packet` structure 26
4.7 Definition of the `mtcp_hdr` structure 27

5.1 Chart of performance on cluster of single-Celeron nodes. 38
5.2 Chart of performance gains on both clusters. 40

List of Tables

2.1 File system calls . . . 7
2.2 Socket system calls . . . 7

3.1 Metasocket messages . . . 18

4.1 Header files . . . 20
4.2 Source files . . . 20

5.1 Process migration pattern used in performance testing. . . . 37
5.2 Comparative performance numbers on the cluser of single-Celeron nodes. . . 38
5.3 Comparative performance numbers on the cluser of dual-Athlon nodes. . . . 40

A.1 Receive flags . . . 50
A.2 Socket options . . . 53

B.1 Socket migration system calls . . . 56

Acknowledgments

I would like to thank Dan Andresen, my parents, and my employers at Geoprobe Systems for their support and patience.

Chapter 1

Introduction

The purpose of this thesis is to discuss a technique for improving network communications of processes running on a MOSIX/OpenMOSIX cluster. This first section explains what MOSIX and OpenMOSIX are, and what the motivations were for this project; the second section details the overall design for my socket migration system; the third section discusses issues encountered during implementation and testing of the project; the fourth section shows the comparative performance of the project's implementation; and the final section lists observations about the project and potential for future work/improvements.

1.1 MOSIX and OpenMosix

MOSIX is a patch to the Linux kernel and a set of userspace utilities that facilitates the automatic distribution of running processes across a cluster as appropriate, its slogan being "fork and forget". MOSIX was initially developed in 1983 by a research team at the The Hebrew University of Jerusalem led by Dr. Amnon Barak. [9] Known at that time as MOS (Multicomputer OS), it was based on Bell Lab's Unix 7 and ran on a cluster of PDP-11 computers. Following revisions were based on Unix System V (1987) [11] and BSD (1993) [7], finally arriving on Linux in 1999. The chief benefit MOSIX provides over a static job-queueing system for a cluster is the ability to dynamically alter the distribution of

processes across a cluster in response to each process's phases execution such as processor- versus network-bound, or being able to move processes off a node if its free RAM is running low to prevent disk thrashing. [8]

In 2001, it was decided that future releases of MOSIX would be proprietary (when it had previously been open source); for commercial use the current MOSIX2 release for the Linux 2.6 kernel is $1000 per node (for the first ten nodes, $50 per node beyond that). [2] Because of this, Moshe Bar began the openMosix fork of MOSIX in February 2002. Moshe Bar announced the end of his involvement in the openMosix project on July 15, 2007, although other people are pursuing further development. [4]

1.2 Origin of Project

The DESPOT project, under the direction of Dr. Andresen, was to develop new scheduling and load-balancing algorithms for processes running on compute-intensive Linux Bewulf clusters. It was decided to use MOSIX as a basis for the project because it provides the infrastructure needed at the kernel level to automatically migrate processes to different nodes in a cluster, and, although it provides its own scheduler for distributing processes across the cluster, it is possible to turn off the builtin scheduler and control the process migration from userspace.

DESPOT's architecture included a master/scheduler node using a plugin-style/DLL interface for the scheduling algorithm to allow experimentation with varying algorithms without having to rebuild the entire program, and a process monitoring tool, *distop,* [6] running on each node to gather information about the memory, processor, and network bandwidth usage of all running processes. A primary goal of DESPOT in comparison to MOSIX was to improve performance by taking into account a process's network usage to more optimally distribute processes during network-intensive phases of execution, which MOSIX didn't do at the time (OpenMOSIX still does not, and MOSIX2 may or may not have improved in

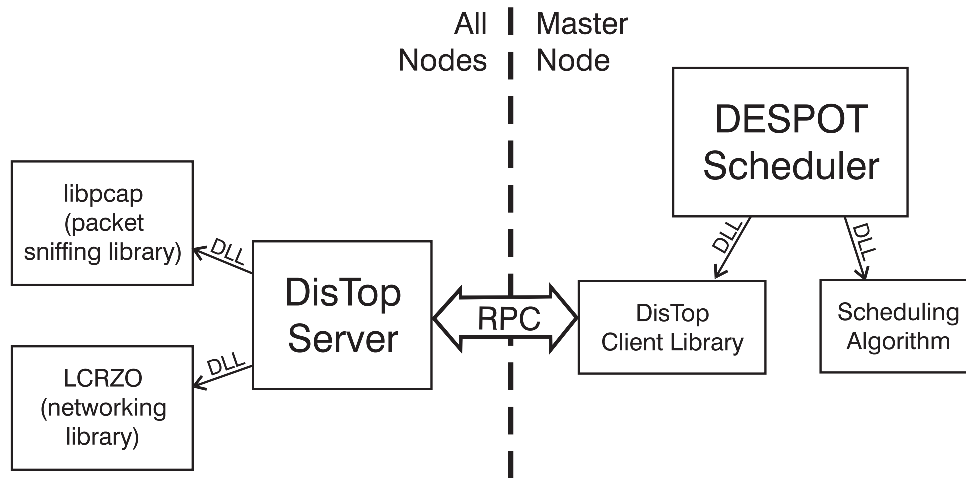

Fig. 1.1. DESPOT system architecture

this regard). A problem with this, though, was that all network communication was routed through a process's starting, or home, node, so that the optimal location for a process with regard to network communication was always its home node, hence the need for migratable network sockets (network connections that would move with the process instead of remaining on its home node). [5]

Chapter 2

TCP/IP and Sockets

The standard implementation of the TCP/IP protocol suite comes from the Berkeley Software Distribution (BSD) of Unix, with the first major release with TCP/IP being version 4.2 released in 1983. This chapter gives a brief overview of TCP/IP and the "sockets" programming interface established by BSD, as well as covering some of the more significant details of TCP specifically. [16]

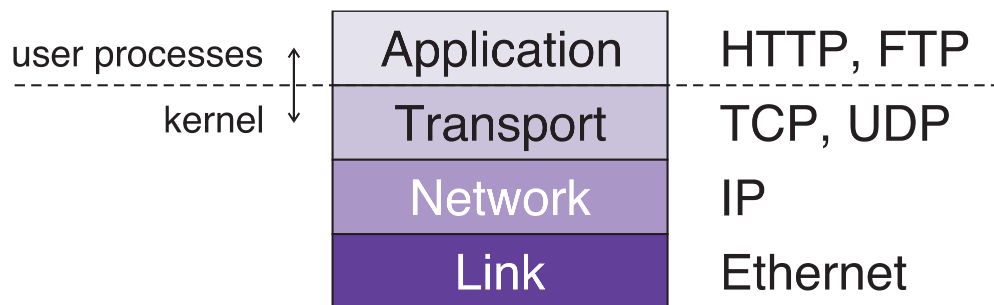

Fig. 2.1. TCP/IP protocol layers [16]

2.1 TCP/IP Protocol Suite

The TCP/IP protocol suite is divided into four layers of communication, shown in figure 2.1: link, network, transport, and application. The *link* layer is the hardware interface, the most common for network endpoints being ethernet. Addressing at the link layer is visible only

to the local network. [16]

The *network* layer is the layer at which hosts on separate networks connected by routers are visible to each other (thus internetwork, or simply "internet"). This functionality is provided by the Internet Protocol (IP). Hosts are identified at this level by 32-bit number (in IPv4, or a 128-bit number in IPv6) referred to as an "IP address" to identify it across all the connected networks.

The *transport* layer is concerned with facilitating an end-to-end connection between specific applications on two hosts. The most common protocols at this level are TCP (transport control protocol) and UDP (user datagram protocol). In both protocols, the application on the host is identified by a 16-bit "port number". Network communications are not a continuous stream of data, but are divided into discrete messages commonly referred to as "packets", which, when transmitted across the Internet, can potentially take different paths and arrive out of order. TCP is concerned with presenting a continuous stream of data to the application and ensuring that the data is correct and is delivered to the application in the correct order, as well as confirming that the remote host has received the data that has been sent.

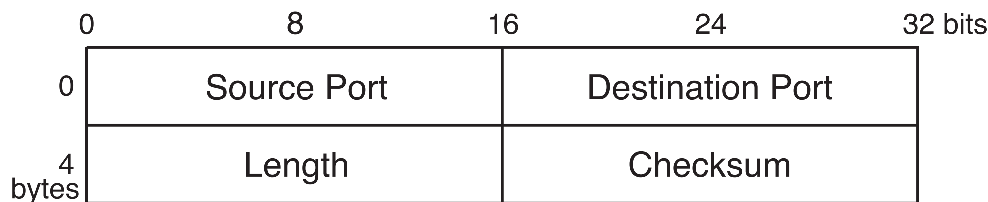

Fig. 2.2. UDP header layout [13]

UDP is a connectionless protocol that presents the data in the packets directly as received. (The "connectionless" means that a single UDP socket is able to send and receive data from multiple sockets instead of being tied to one other socket with which to communicate.) UDP does little to guarantee delivery or ordering, although it can optionally have a checksum in the packet header to verify that the data in that packet is at least correct. These characteristics make it a much simpler protocol than TCP with a lower overhead, both in processor use and

header size.

Finally, at the highest level, is the *application* layer. As the name suggests, this is the specific application running on a given host, such as, for instance, a web browser or web server. While all lower-level protocols are implemented in the kernel and/or hardware, application-level are user space programs.

2.2 Berkeley Sockets

The most common application programming interface (API) for software that makes use of the TCP/IP protocols is the *sockets* interface, sometimes referred to as *Berkeley sockets* because it originated in 4.2 BSD. Although TCP/IP is the primary concern of this paper, the sockets interface is not limited to the TCP/IP protocols. Another protocol suite accessible through the sockets interface is the set of Unix domain protocols. These protocols can be used for interprocess communication on the same host, using the same programming interface but not having to go through the lower layers of the network stack for increased efficiency. [17]

The sockets interface consists of a set of system calls (kernel routines that are callable from user applications). Sockets are treated the same as open files, so that, while there are socket-specific system calls (see table 2.2), system calls for file I/O (see table 2.1) are also applicable to sockets. An open file (or socket) in a process is referenced by a number which is selected by the kernel when the file/socket is opened or created, called a file descriptor.

Sockets generally follow a client/server model, where the server is the side that listens for connections, and the client is the side that attempts to connect to the server. Generally speaking, for TCP and UDP sockets, the general defining characteristics by which a socket can be accessed are the IP address of the host the socket is on and the port number it is bound to, as well as the IP address and port number for the remote socket if it is connected. All sockets are initially created through the `socket` call, which does not bind the socket to a particular port or make a connection, but only sets the protocol family and type of socket, the

Category	Name	Function
input	`read` `readv`	receive data into a single buffer receive data into multiple buffers
output	`write` `writev`	send data from a single buffer send data from multiple buffers
I/O	`select`	wait for I/O conditions
termination	`close`	terminate connection and release socket
administration	`fcntl` `ioctl`	modify I/O semantics miscellaneous socket operations

Table 2.1. General file system calls that apply to sockets

Category	Name	Function

setup	`socket`
	
	`bind`
		create a new unnamed socket within a specified communication domain
		assign a local address to a socket

server	`listen` `accept`	prepare a socket to accept incoming connections wait for and accept connections
client	`connect`	establish a connection to a foreign socket

input	`recv`
	`recvfrom`
	`recvmsg`
		receive data specifying options
		receive data and address of sender
		receive data into multiple buffers, control information, and receive the address of sender; specify receive options

output	`send`
	`sendto`
	`sendmsg`
		send data specifying options
		send data to a specified address
		send data from multiple buffers and control information to a specified address; specify send options

termination	`shutdown`	terminate connection in one or both directions

administration	`setsockopt`
	`getsockopt`
	`getsockname`
	`getpeername`
		set socket or protocol options
		get socket or protocol options
		get local address assigned to socket
		get foreign address assigned to socket

Table 2.2. Socket-specific system calls

two types of interest in this context being stream-oriented (e.g. TCP) and datagram-oriented (e.g. UDP) sockets.

In order to prepare a socket to listen to a particular port as a server, it is necessary to call the `bind` (to bind the socket to a specific port by which the client can connect to it) and `listen` system calls. Then, the `accept` call is used to actually accept an incoming connection, which creates a new socket for the connection, leaving the original socket free to continue listening for new connections. Initiating a connection from the client side is much simpler, needing only to call the `connect` system call, which selects the local port for the socket automatically. [18]

2.3 TCP

The original TCP specification is documented in RFC 793 [14]. A central concept of TCP is that of "sequence numbers", which are used for ordering, delivery confirmation, and flow control. For each direction (server to client and client to server), each byte of the transmission also has a contiguously-ordered sequence number, with the starting value chosen to avoid duplicate sequence numbers for the same connection on the network. By including a packet's starting sequence number in the TCP header (see figure 2.3) it is possible to determine if the packet is a duplicate or if it has arrived out of order. For delivery confirmation, an "Acknowledgment Number" is included in the header that, if the appropriate flag is set, contains the expected starting sequence number for the next packet. The acknowledgment mechanism allows the networking code to determine if a packet hasn't been received (and acknowledged) in a reasonable amount of time and needs to be retransmitted. Finally, for flow control there is a "Window" field in the packet header to indicate how far into sequence space (essentially how many bytes, with the exceptions noted below) unacknowledged data will be received. In other words, if the transmitter sends the amount of data indicated by the window field before an acknowledgment is received, further packets will be dropped.

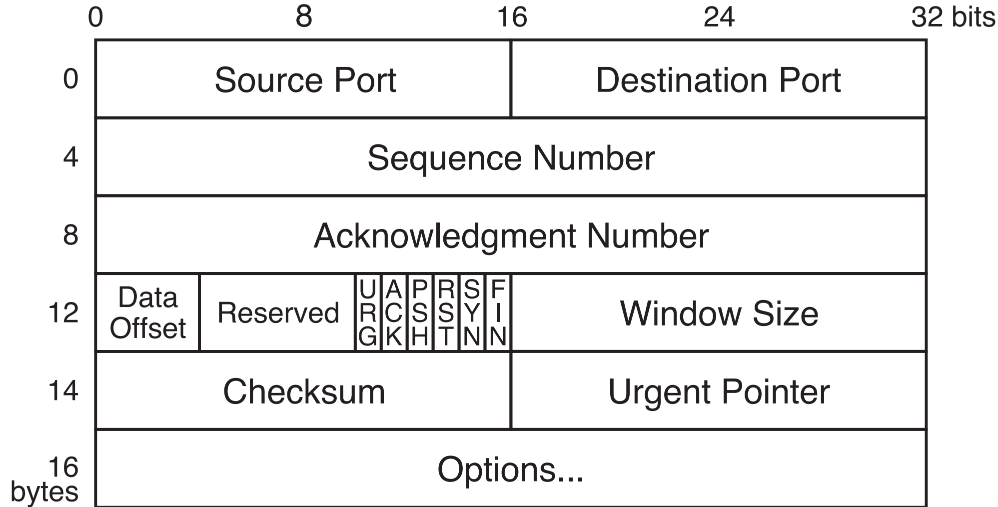

Fig. 2.3. TCP header [14]

The header contains six flags that indicate how the packet should be interpreted. The most notable of these for the purposes of this paper are ACK, RST, SYN, and FIN. The SYN (synchronize) flag is used in establishing the connection, and the FIN (finish) flag is used to request that the connection be terminated. The RST (reset) flag is used to immediately reset the connection or indicate that the packet just received was invalid. The ACK (acknowledge) flag is used to indicate an acknowledgment of received data, in conjunction with the "Acknowledgment Number" field. The PSH (push) flag was originally intended to signal that the data in the current and previous packets should be delivered immediately to the user application, but most socket implementations don't delay delivery of data to the user application, so this flag is generally ignored. [16] Finally, the URG (urgent) flag, in conjunction with the "Urgent Pointer" field facilitates a secondary stream of data being transmitted on the same socket, referred to in the TCP specification as "urgent data" or, in the BSD sockets documentation as "out-of-band data".

The process of establishing a TCP connection is commonly referred to as a "three-way handshake", because three packets must be transmitted and received (two by the client and one by the server) before the connection is fully established. The purpose of the SYN flag (short for "synchronize") is to set the starting sequence number for the transmitting direction

of the connection, and a packet with the SYN flag set with no other flags set is treated as a connection request from a (potential) client. The SYN and FIN flags both occupy one value in sequence space (SYN being treated as falling at the beginning of the packet in sequence space, and FIN being treated as coming at the end), so the proper response of a server accepting the connection is to reply with the SYN and ACK flags set, the acknowledgment number for the packet being one higher than the sequence number for the first packet. Finally, to complete the connection, the client sends back an acknowledgment of the second packet. This is illustrated in figure 2.4.

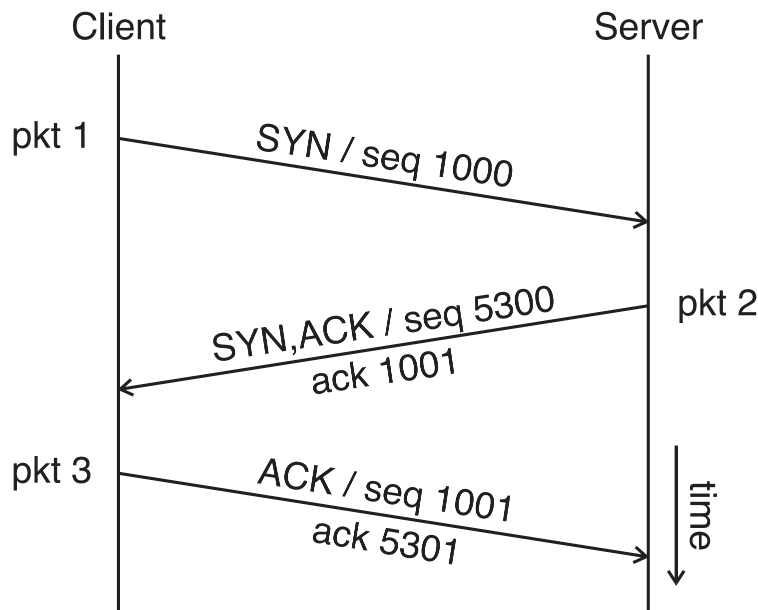

Fig. 2.4. An example of the TCP connection "three-way handshake"

The TCP specification defines eleven valid states for a socket, shown in figure 2.5, which will not be discussed in detail here. These states are Closed, Listen, SYN Received, SYN Sent, Established, FIN Wait 1, FIN Wait 2, Closing, Time Wait, Close Wait, Last ACK, and Closed.

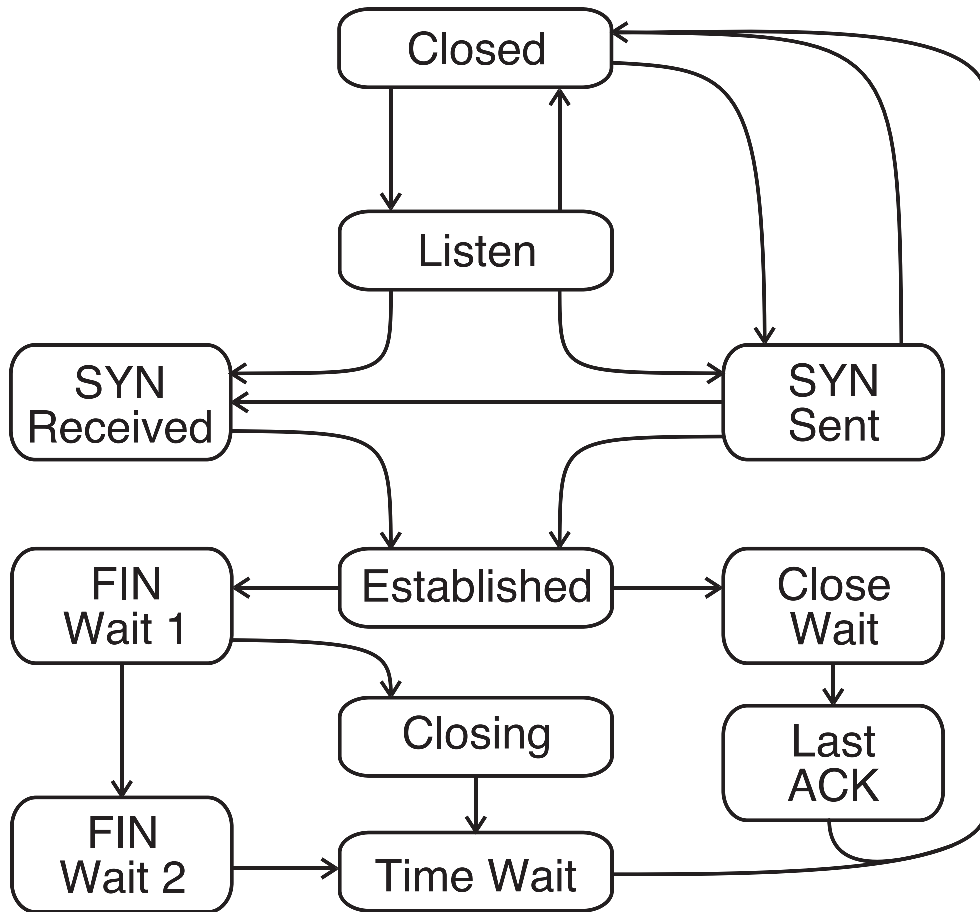

Fig. 2.5. TCP states [14]

Chapter 3

Design

This chapter discusses the general design of the socket migration system put forth in this thesis, as well as motivations for the design.

3.1 Design Goals

In the context of a clustering system such as MOSIX or OpenMOSIX, the primary goal of introducing a socket migration feature would be to reduce or eliminate the performance hit a process takes in network communication when it is away from its home node. Here are other design targets for the socket migration system, most of which have been fulfilled:

Transparency. Ideally, applications should not need to be rewritten to take advantage of socket migration system. Unfortunately, though, the project was later restricted in scope such that currently separate, still largely BSD-style, system calls are required to use the sockets in this system.

Kernel-Level Support. For performance reasons, socket migration should be implemented at the kernel level and automatically follow process migrations.

Portability. The system should be relatively self-contained so as to be easily ported to future kernel versions or different process migration systems.

Interoperability. Nodes in the cluster should still be able to communicate with hosts outside the cluster.

State Preservation. The socket must retain its state when migrated.

Delivery of All Packets. If any packets were pending during the migration, none of them must be discarded.

Allowance of Simultaneous Migration. The connection between processes should still be correctly maintained even when both processes migrate to new nodes simultaneously.

Unique Addressability. The sockets should be uniquely addressable within the cluster (i.e. processes with different home nodes should be able to use the same ports and not interfere with each other even if they migrate to the same node).

Migratability of Server Sockets. Socket migration should not be limited to sockets that are already connected.

3.2 Related Work

This thesis project is built on MOSIX/OpenMosix, which was mentioned in section 1.1. The last stable version of OpenMOSIX is still based on the 2.4.26 version of the Linux kernel. A port to the 2.6 kernel has been in progress for a while now, and this unstable version still doesn't have any kind of socket migration capability. MOSIX2 was announced in August 2007 and is the "official" MOSIX for the 2.6 kernel. The primary new feature mentioned in the white paper [10] is the ability to coordinate multiple clusters e.g. across a campus network. The white paper goes into little detail regarding MOSIX2's socket migration facilities, describing the system as involving each process running on the cluster being assigned a "mailbox". Said description most likely indicates that making use of socket migration in MOSIX2 involves writing (or recompiling) user applications to target a different socket type.

Another clustering system for Linux is OpenSSI (Single System Image), which has an emphasis on making the cluster appear as a single computer to the user, including a distributed filesystem, unique process IDs across the entire cluster, among other features. Much like MOSIX, a significant portion of its functionality is implemented in the kernel, but, to achieve the full single system image effect, changes are made to a number of userlevel utilities, making installation more involved if one wants to run it on a distribution of Linux for which it has not been packaged. Its development appears to be slightly more active than that of OpenMOSIX, but like OpenMOSIX it still has no socket migration ability the stable version is still stuck on the 2.4 branch of the Linux kernel. [3]

One more clustering system to mention is Kerrighed. Somewhat less mature than the others mentioned, its development began in 1999 at the French research facility INRIA. Nonetheless, it is currently under quite active development, having had several significant releases since April. Kerrighed apparently has some features that would go a long way toward transparent cluster operation, although some of them have been temporarily disabled in the process of porting the system to the 2.6 branch of the Linux kernel, namely distributed inter-process communication (pipes, local and TCP sockets), and distributed threads. [1]

A related project to mention is a standalone socket migration solution undertaken by students at CMU in 2002 called MIGSOCK, designed to be independent of the particular clustering/process migration system being used. The actual mechanism for achieving the migration is in the kernel, predominantly in a kernel module, but still with some modifications to the kernel itself. The migration of the sockets, though, is controlled by a usermode API, which involved copying the socket information to a user buffer, and, as it was tested, saving it to a shared filesystem, then, on the new node, creating a new socket for the migrated process through the `socket()` system call and calling the MIGSOCK API to restore the socket state. The implementation of this thesis aimed to improve over this project in three ways: allowing multiple sockets to be connected on the same ports as long they didn't start on the same node; allowing communication to resume correctly even if both of the

communicating processes migrate at the same time; and in making the socket migration process entirely automatic within the kernel instead of needing to be controlled by a user process. [12]

3.3 Practical Considerations

This thesis project is chiefly concerned with the transport layer of the TCP/IP protocol stack, which, as mentioned in chapter 2, consists primarily of UDP and TCP. UDP is the simpler of the two protocols, being connectionless and a fairly thin layer over the underlying IP (Internet Protocol) data packets. It does not generally make any guarantees regarding the correctness of the packets or the order in which they are delivered. TCP is the “heavier” of the two protocols, being strongly oriented around a connection between two endpoints and providing a stream of data derived from the delivered packets, which are guaranteed to be both the correct data and in the correct order independent of the order in which the packets were received; and providing confirmation to the sender that the data was successfully delivered by way of acknowledgment from the host on the receiving end. [cite RFCs 793 and 768]

Some of the more pragmatically-motivated design choices arise from the differences between the two protocols. Since UDP is so open-ended, migration of UDP sockets would be more difficult and have greater overhead, quite possibly requiring the location of all UDP sockets to updated across the entire cluster, so for this project I decided to limit myself to TCP (or at least a TCP-like protocol; I discuss this further in section 3.4) sockets.

For the sake of portability, as mentioned in section 3.1, packets for the migratable sockets are tunnelled over the existing IP stack to avoid changes to the existing networking code in the kernel, for the most part. Although the differences between TCP and UDP mean that UDP sockets take more work to migrate, they also mean that UDP works well as a thin layer to build on, especially since this socket migration implementation provides (most of)

the features of TCP itself and doesn't need to duplicate the overhead of the protocol, so that is the protocol over which the packets are tunnelled.

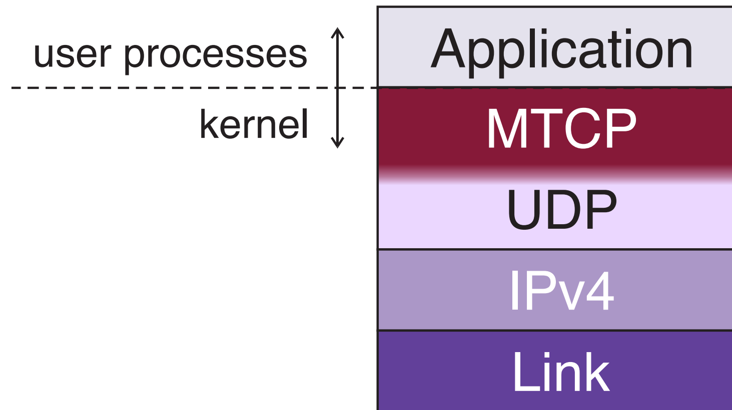

Fig. 3.1. MTCP protocol layering

3.4 MTCP

In order to satisfy the requirement of sockets within the cluster being uniquely addressable within the cluster (see section 3.1), some additions to the TCP packet header are needed. The "new" version of the protocol with larger packet header is referred to as as MTCP (Migratable Transmission Control Protocol). The new fields in the MTCP header are the home nodes of the receiving and sending processes, as shown in figure 3.2.

Although MTCP is essentially an implementation of TCP according to the specification, some of TCP's features have not been omitted to limit the scope of the project. The omissions/simplifications are using a fixed send/receive window, no support for urgent/out-of-band data, and no support for any options, with the exception of using the option field to signal migration.

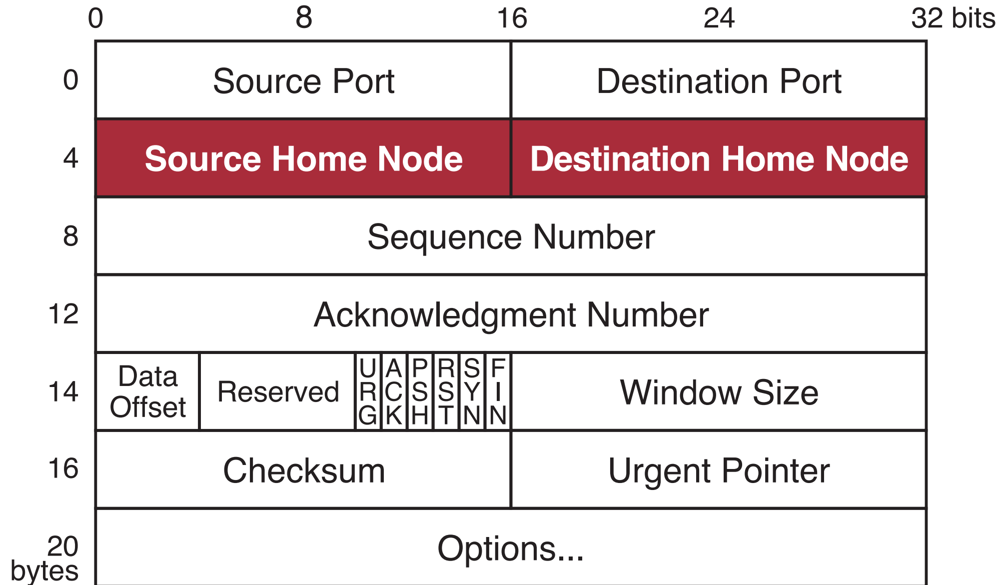

Fig. 3.2. MTCP header layout. Fields not in the original TCP header are shown in bold face.

3.5 Metasockets

To facilitate socket migration, an additional channel of communication is needed for the purpose of transmitting information about migrating sockets between nodes. This requirement is fulfilled by node-to-node fixed (i.e. non-migrating) sockets referred to as "metasockets". The metasockets, although non-migrating, use the MTCP socket functionality, and are indicated to be metasockets by setting the values of the ports and home nodes in the packet header to 0.

Of the messages listed in table 3.1, the most important is `MIGTOYOU`, as it is the one into which the socket information is serialized. Also, any packets that were pending to be sent over the socket or delivered to the process are included in this message.

Use of the `MIGACKED` message allows both processes to migrate simultaneously and still have both ends of the socket migrate successfully, because the socket information can be retained on the previous node until the other end of the connection has acknowledged the migration.

Information on a process's server sockets is retained on the process's home node even

Title	**Description**
MIGTOYOU	Sent to the node a process is migrating to.
MIGACKED	Sent to the node a process has migrated from when the remote process has acknowledged the migration, such that the old socket entry is no longer necessary.
MIGLISTEN	Sent to a process's home node when the process migrates if the process has any listening (server) sockets open.
OPENLISTEN	Sent to a process's home node when that process opens a listening (server) socket.
CLOSELISTEN	Sent to a process's home node when that process closes a listening (server) socket.

Table 3.1. Metasocket messages

when that process has migrated so that a process wishing to connect can send a connection request to the process's home node and find out where to redirect its request if the process has migrated away. Keeping this information up-to-date is facilitated by the `MIGLISTEN`, `OPENLISTEN`, and `CLOSELISTEN` messages.

Chapter 4

Implementation

This chapter details the key elements of the socket migration implementation put forth in this thesis.

4.1 Organization

The code for the migratable socket implementation exists as a set of additions and modifications to version 2.4.26 of Linux kernel, which is the last version for which a stable version of openMosix has been released. The main MOSIX/openMosix code is located in the directory `hpc` in the kernel source tree, with header files being stored in the directory `include/hpc`. Since the project is generally an extension of openMosix, the main source code and header files are in a directory `migsock` under the `hpc` and `include/hpc` directories, respectively. The header and source files specific to this project are listed in tables 4.1 and 4.2, respectively.

Additionally, the project also includes some small modifications to standard kernel and OpenMosix source files. The added user system call is facilitated by changes to `entry.S` and `mosasm.H` in `arch/i386/kernel`, as well as `hpc/syscalls.c` and `include/asm-i386/unistd.h`. A new call in `hpc/config.c` initializes the migratable sockets at the same time as openMosix is initizialized. Hooks in the files `mig.c` and `remote.c` in the directory `hpc` notify the socket migration code when a process is migrating. Finally, a couple of additional macros

Filename	Description
`migsock.h`	Function and structure declarations for use from other parts of the kernel.
`ms_internal.h`	Function, structure, etc. declarations for use by code in the `hpc/migsock` directory.
`ms_debug.h`	Declarations of debugging macros.
`ms_systest.h`	Definitions of system call types for migratable sockets.

Table 4.1. Header files, stored in `include/hpc/migsock`.

Filename	Description
`ms_systest.c`	Migratable sockets system call functions.
`ms_external.c`	Functions composing the API to the rest of the kernel.
`ms_data.c`	Functions for creating, deleting, referencing, and accessing sockets.
`ms_pkt.c`	Functions for creating, deleting, and manipulating packets. (This includes reading from and writing to a socket.)
`ms_routing.c`	Functions for handling and responding to all packets coming into a socket.
`ms_input.c`	Functions for handling packets coming from the network.
`ms_sockopts.c`	Functions for setting and retrieving socket options.
`ms_metasock.c`	Metasocket routines.
`ms_migd.c`	Daemon for responding to process migration.
`ms_debug.c`	Debugging routines.

Table 4.2. Source files, stored in `hpc/migsock`.

in include/linux/sched.h are used to find a process on the current node based on its home node and the process ID it had on that node, as unique process IDs are not maintained across the cluster. Also, inclusion of the migratable socket code into a compiled kernel is controlled by several new options in the kernel build system added to the file arch/i386/config.in.

4.2 Sockets

As noted in table 4.2, general socket maintenance is handled in ms_data.c. This includes creation and deletion (ms_create_sock() and ms_delete_sock()), system-wide socket hash table maintenance (ms_init_sock_tables(), ms_hash_sock(), and ms_unhash_sock()), searching for sockets (ms_find_sock_port(), ms_find_sock_fd(), and ms_find_metasock()), and socket locking (ms_acquire_slock() and ms_release_slock()). The socket structure, struct mig_sock, is defined in ms_internal.h. Its fields can be divided into the following categories:

- Local and remote connection information
- Socket status
- Sequence variables (mostly from section 3.2 of RFC 793)
- Synchronization and threading variables
- Packet queues
- Socket list/list entries
- Resend variables
- Socket options
- Fields used to facilitate migration

```
struct mig_sock {
        int fd;
        pid_t pid;
        int local_hnode;
        int local_cnode;                                                    5
        int rem_hnode;
        int rem_cnode;
        __u16 local_port;
        __u16 rem_port;
        struct sockaddr_in rem_cip;                                         10

        ...
};
```

Fig. 4.1. The connection fields of the socket structure

The connection fields of the socket, shown in figure 4.1, are as follows: `fd` is the file descriptor; `pid` is process ID (on the process's home node) of the process that owns the socket; `local_hnode` and `rem_hnode` are the home nodes of the local and remote processes, respectively; `local_cnode` and `rem_cnode` are the current nodes of the local and remote processes respectively; `local_port` and `rem_port` are the ports of the local and remote ends of the connection; and `rem_cip` is a cache of the current IP address of the process on the remote end of the connection.

The status fields are listed in figure 4.2 along with any value definitions for those fields that are defined in my code. The values of the `mts_state` enumeration used by the `state` field correspond directly to the states listed in section 3.2 of RFC 793. The `iostate` field indicates if the socket can currently be read from or written to. It is defined as a bitfield of the `MTIO_RD` and `MTIO_WR` values, for reading and writing respectively, which also happen to match the values used by the `how` parameter of the `shutdown()` system call. If an error occurs, causing the socket to be disconnected, the value of that error, as defined in `errno.h`, is stored in the `errno` field until it is able to be received by the process that owns the socket.

The sequence variables listed in figure 4.3 mostly correspond to the send and receive sequence variables listed in section 3.2 of RFC 793 [14], although some of them are not currently

```
enum mts_state {
        MTS_CLOSED=0, MTS_LISTEN, MTS_SYN_SENT, MTS_SYN_RECEIVED,
        MTS_ESTABLISHED, MTS_FIN_WAIT_1, MTS_FIN_WAIT_2, MTS_CLOSE_WAIT,
        MTS_CLOSING, MTS_LAST_ACK, MTS_TIME_WAIT
};                                                                      5

#define MTIO_RD 1
#define MTIO_WR 2

struct mig_sock {                                                       10
        ...

        enum mts_state state;
        int iostate;
        int newly_migrated;                                             15
        long errno;

        ...
};
```

Fig. 4.2. The status fields of the socket structure, along with corresponding value definitions

```
struct mig_sock {
        ...

        __u32 snd_una;
        __u32 snd_nxt;                                                  5
        __u32 snd_wnd;
        __u32 snd_up;
        __u32 snd_wl1;
        __u32 snd_wl2;
        __u32 iss;                                                      10
        __u32 rcv_nxt;
        __u32 rcv_wnd;
        __u32 rcv_up;
        __u32 irs;
        __u32 deliver_nxt;                                              15
        __u32 prev_ack_sent;

        ...
};
```

Fig. 4.3. The sequence variables for the socket structure

in use due to the lack of support for urgent/out-of-band data (snd_up and rcv_up) or adaptive send/receive window sizes (snd_wl1 and snd_wl2). The two fields that don't correspond to any of the sequence variables listed in RFC 793 are deliver_nxt and prev_ack_sent. The deliver_nxt field contains the sequence number of the next byte to return when a process reads from the socket, allowing for reading partial packets, and prev_ack_sent is used by the socket's resend thread to keep track of when it needs to send a new acknowledgement.

```
struct mig_sock {
        ...

        struct semaphore lock;
        pid_t lock_pid;                                                5
        atomic_t lock_cnt;
        struct task_struct *rt_thread;
        wait_queue_head_t wait_queue;

        ...                                                            10
};
```

Fig. 4.4. Sychronization and threading variables for the socket structure

The synchronization and threading variables of the socket structure are listed in figure 4.4. The lock field is, as the name implies, the socket's lock. When a socket is connected to another process on the same node, updates to the other socket are made directly from the same socket, necessitating recursive locks (i.e. allowing the same process to acquire the same lock more than once). Semaphores in the Linux kernel do not natively have this capability [15], so some additional structure is required, namely, the lock_pid and lock_cnt fields. The lock_pid field is the ID of the process that currently holds the lock, and lock_cnt is the number of times the lock is currently held. So, if a process fails to acquire the lock immediately, but its process ID matches lock_pid, it simply increments lock_cnt and continues on its way; otherwise, it sleeps on the lock. The wait_queue field is for processes waiting for something to happen to the socket, such as the owner of the socket waiting to receive data. The rt_thread field is a pointer to the process information

for socket's retransmission thread, which also handles other time-based socket maintenance.

```
struct mig_sock {
        ...

        struct list_head connect_queue;
        struct list_head list_port;                                         5
        struct list_head list_connect;

        ...
};
                                                                             10
struct ms_task {
        pid_t pid;
        int cnode, hnode;
        int prev_node;
        int socks_count;                                                    15
        struct mig_sock *socks[SOCKS_PER_PROCESS];
        struct list_head list;
};
```

Fig. 4.5. Socket list/list entry fields in the socket structure

Sockets are referenced in several ways. Global hash tables are maintained for sockets by ports & home nodes, and by process; and server sockets maintain a list of pending connections. The `connect_queue` field shown in figure 4.5 is head node for the list of pending connections if the socket is a server socket, and, if the socket is a pending connection, `list_connect` is its entry in the parent socket's connection queue. The `list_port` is the socket's entry in the hash table that is sorted by port. (The only value used for hashing is the local port, but of course when searching for a socket, remote port and local & remote home nodes are checked as well.)

In the initial implementation, sockets were directly hashed by file descriptor, which would result in hash collisions since file descriptors are not unique across multiple processes. Now, there is a hash table for process information, including the sockets that process currently owns. This information is stored in the `ms_task` structure, also shown in figure 4.5. Pointers to a process's sockets are stored in a simple array `socks`, and `socks_count` contains the

number sockets owned by the process.

4.3 Packets

4.3.1 Structures

```
struct ms_packet
{
        struct sockaddr_in addr;
        int datalen;
        int seqlen;                                             5
        struct msghdr msg;
        struct sk_buff *skb;
        unsigned long timestamp;
        struct list_head list;
};                                                              10
```

Fig. 4.6. Definition of the `ms_packet` structure

The important packet-related structures, namely `ms_packet` and `mtcp_hdr`, are defined in `ms_internal.h`. As shown in figure 4.6, the actual packet data is pointed to by an embedded `msghdr` structure (the same structure used by the `sendmsg()` and `recvmsg()` system calls), field `msg`. The socket address is embedded in the structure (field `addr`) since the the `msghdr` structure contains a pointer to the address, and not the address structure itself. The `datalen` field is the length of the packet data pointed to from within the `msg` field, including the packet header. The `seqlen` field contains the TCP sequence length of the packet. This means the length of the packet's content in bytes, plus 1 each for the SYN and FIN flags in the header if either of those are set. If the packet in question came from the network, or the packet data is otherwise contained in a `sk_buff` structure (the kernel's existing packet structure), then the `skb` field is a pointer to that structure. Otherwise, it is a null pointer. The `timestamp` field contains the time of the last attempt to send the packet, using the kernel clock in the global variable `jiffies`. Finally, the `list` field is the packet's

```
struct mtcp_hdr
{
        __u16 sport;
        __u16 dport;
        __u16 shnode;                                        5
        __u16 dhnode;
        __u32 seqnum;
        __u32 acknum;

        unsigned int reserved1:4;                            10
        unsigned int dofs:4;
        unsigned int fin:1;
        unsigned int syn:1;
        unsigned int rst:1;
        unsigned int psh:1;                                  15
        unsigned int ack:1;
        unsigned int urg:1;
        unsigned int reserved2:2;

        __u16 window;                                        20
        __u16 cksum;
        __u16 urgptr;
        char options[0];
};
```

Fig. 4.7. Definition of the `mtcp_hdr` structure

entry in a packet queue.

The `mtcp_hdr` structure, shown in figure 4.7, is the "MTCP" header structure and corresponds directly to the header layout as shown in figure 3.2 on page 17. The `sport` and `dport` fields are the source and destination ports of the two sockets, respectively. The `shnode` and `dhnode` fields are the home nodes of the source and destination processes, respectively. The `seqnum` field is the starting sequence number of the packet, and, if the ACK flag is set (field `ack`), the `acknum` field contains the expected starting sequence number of the next incoming packet. The field `dofs` is the "Data Offset" field, indicating the length of the header in bytes divided by four. The 1-bit fields (indicated by a `:1` after the field name) are header header flags. In the current implementation, the PSH and URG flags are unused. SYN ("synchronize") is used when establishing the connection to set the initial sequence number,

FIN ("finish") is used in terminating a connection, and a packet with the RST ("reset") flag set is sent in response to an invalid packet. The `window` field is the sequence length of the receive window, the `cksum` is the packet's checksum, the `urgptr` field is unused in the current implementation. The zero-length array `option` is a convenience pointer to any option fields that may come after the header.

4.3.2 Lower-Level Routines

What I would call the "lower-level" packet creation, manipulation, etc. routines are in the file `ms_pkt.c`. There are two functions relating to packet creation, one of which is deprecated. Originally, when I was sending packets on the network with a lower-level version of the `sendmsg()` system call (`sock_sendmsg()`), which need a `msghdr` structure, which would be created from a set of buffers by `ms_create_pkt()`. The new packet-creation routine, `ms_alloc_pkt()`, allocates memory for the packet as the kernel's internal packet structure, `sk_buff` and populates the `ms_packet` structure with pointers into the `sk_buff`. This is discussed further in section 4.4. The `ms_free_pkt()` function deallocates a packet that is no longer needed.

Packets are transmitted by the `ms_send_pkt()` and `ms_resend_pkt()` functions, with `ms_resend_pkt()` being used to send a packet that has already been sent once before. If the packet is being sent to a local socket, these functions pass the packet to `ms_route_pkt()`, removing the steps of populating and verifying the checksum field of the packet header and going through the lower-level network stack. How these functions handle transmitting packets across the network is covered in section 4.4.

Sending and receiving data from user applications are handled at the lower level by the `ms__sendmsg()` and `ms__recvmsg()` functions, respectively. The `ms_cksum_calc()` function calculates a packet's checksum either to put into the header or to compare against the checksum in the header in a received packet, and, finally, the `ms_crop_pkt()` function, as the name implies, crops a packet.

4.3.3 Routing

The two "top-level" functions in `ms_routing.c` are `ms_route_pkt()` and `ms_sock_thread()`. The function `ms_route_pkt()` processes an incoming packet (after the checksum has been verified, if necessary), determining what socket it goes to and/or how to respond to the packet. This is done as specified by the TCP specification, with the exception of the short-cuts noted earlier, and some additional checks for socket migration that are discussed in section 4.5.

For each connected or almost-connected socket, a background thread is created that runs the function `ms_sock_thread()`. The `ms_sock_thread()` function handles sending standalone acknowledgment (ACK) packets, resending packets if necessary, and sending packets that are placed on the socket's send queue. (Packets to send are only queued if a send is attempted before the socket has finished connecting.)

4.4 Interface to Existing IP Stack

The key portions of the interface to the existing network code in the kernel are in `ms_input.c` (handling packets incoming from the network) and `ms_pkt.c` (allocating lower-level packet structure `sk_buf`, and transmitting packets on the network). The primary function in `ms_input.c` is `ms_recv_daemon()`, which is launched as a separate thread on initialization and is responsible for creating the lower-level UDP socket used to send and receive packets and for handling incoming packets. When a new packet arrives, it is passed to another thread that creates an `ms_packet` structure for the packet and verifies that the packet has come from within the cluster and that the checksum is matches the packet, then passes the packet on to `ms_route_pkt()`. (When both sockets are on the same node, the checksum field in the header is not populated or verified.)

As noted in section 4.3.2, the newer packet-creation routine `ms_alloc_pkt()` allocates a lower-level packet buffer then creates an `ms_packet` structure to point to it. In both

ms_send_pkt() and ms_resend_pkt(), if the packet's sk_buff pointer skb is null, it calls sock_sendmsg(), the lower level function that is called by the sendmsg() system call. To do this checking for the pointer not to be in kernel memory space is temporarily disabled since a buffer in user memory is expected. This results in two copies of the data to transmit being made every time, first from user space into kernel space, and then a second time by sock_sendmsg(). To improve performance in this area, I added the ms_alloc_pkt() function and wrote the function ms_send_pkt_skb() to send the packet more directly from my own code. If the packet is being retransmitted then the data still needs to be copied (hence the need for the ms_resend_pkt() function), but this at least optimizes the common case. At this point most of the signalling and acknowledgment code still uses the old ms_create_pkt() function but ms__sendmsg() and ms__recvsmsg() were rewritten to take advantage of the new functions so that most of the data transmitted on the socket is optimized in this way.

4.5 Socket Migration

The bulk of the socket migration code is in the files ms_metasock.c and ms_migd.c, although there are also related bits in ms_external.c, ms_data.c, and ms_routing.c. All metasocket processing is in ms_metasock.c, and ms_migd.c contains a daemon that handles properly responding to process migration. This is necessary because attempting to perform the necessary allocations, etc. directly from the context of the migration cause problems with the migration process. Secondarily, ms_external.c contains the means of accessing the code in ms_migd.c from the rest of the kernel and locks a process from migrating while it is establishing a connection; ms_data.c contains some code for handling an internal representation of processes; and ms_routing.c handles the interaction between a migrating socket and the socket on the other end of the connection.

4.5.1 Migration Daemon

In `ms_migd.c`, the migration daemon is accessed by the `ms_signal_migd_outbound()` and `ms_signal_migd_inbound()` functions, for processes migrating onto or away from the node, respectively. These functions are called from the `ms_process_migrate_outbound()` and `ms_process_migrate_inbound()` functions in `ms_external.c`, mentioned in section A.

The daemon itself is contained in the function `ms_migd_thread()`, which, as appropriate, calls functions `ms_migd_handle_outbound()` and `ms_migd_handle_inbound()`. From here, inbound process handling is the simpler of the two cases, simply adding some internal information about the process (what node it came from, etc.) and locks the process from migrating away if connected sockets for the process have already migrated onto the node. (A process with connected sockets is locked from migrating until the remote sockets have acknowledged the migration.) For outbound processes, internal process information is also appropriately updated, as well as the `local_cnode` field of all sockets owned by the process, the metasocket message MIGTOYOU is sent to the new node (if the process has any sockets), the MIGLISTEN message is sent to the home node for any server sockets, and on a timeout (currently 30 seconds) the server sockets are deleted.

4.5.2 Metasockets

In `ms_metasock.c`, the primary means of accessing the metasocket functionality is the function `msm_send_msg()`, which accepts the message types listed in table 3.1 on page 18. The actual metasocket functionality is actually accomplished in various background threads: listening for connection requests (`msm_listen_thread()`), receiving messages (`msm_accept_thread()` and `msm_recv_thread()`), sending messages (`msm_connect_thread()` and `msm_send_thread()`), and processing messages (`msm_proc_thread()`). Which functions are used for sending and receiving messages whether the current node initiated the connection— `msm_accept_thread()` and `msm_send_thread` if initiated by the other node, otherwise `msm_recv_thread()` and `msm_connect_thread()`. In both cases, the lower-level functions called

for sending and receiving are msm_send_loop() and msm_recv_loop(), respectively. Both of these functions watch for either incoming messages or requests for new messages to send then dispatch as appropriate. In the case of sending, the function msm_build_and_send_msg() called, and for a message that is received it is passed on to the processing thread (a new one is launched if one is not already running). Process migration messages are built by msm_build_migtoyou_msg(), and all others are built by msm_build_connectinfo_msg(). Message structures are populated from incoming packets by msm_parse_incoming_msg() and msm_parse_migtoyou_msg().

Each metasocket message type has its own function for appropriately handling the message. These are msm_proc_migtoyou_msg(), msm_proc_migacked_msg(), msm_proc_miglisten_msg(), msm_proc_openlisten_msg(), and msm_proc_closelisten_msg() for MIGTOYOU, MIGACKED, MIGLISTEN, OPENLISTEN, and CLOSELISTEN, respectively.

As one might guess, the most involved message type to handle is the MIGTOYOU. This involves allocating memory for the recreations of the sockets, copying the snapshots of the sockets over, clearing and/or intializing any fields that do not transfer over directly (pointers, locks, etc.), rebuilding any packet queues that might not have been empty, starting retransmission threads, linking the sockets into system-level hash tables, and, if any of the sockets are established connections (and the process has already arrived on the node), locking the process from migrating. (The process is allowed to migrate again once all remote sockets have acknowledged the migration.)

The MIGACKED message is sent to a socket's previous node when the remote end of the connection has acknowledged the migration. At this point the old copy of the socket structure is deleted from the node. (The socket is retained so that ms_route_pkt() can respond with a migration signal if the other end of the connection is unaware of the migration.)

The MIGLISTEN, OPENLISTEN, and CLOSELISTEN messages are all sent to a process's home node to report changes in a server socket. The MIGLISTEN message indicates that the process has migrated to a new node and the specified socket's should local_cnode

field should be updated with the process's new node. The OPENLISTEN and CLOSELISTEN messages indicate that the specified socket should be created (with the `local_cnode` initialized to the process's new current node) or deleted, respectively.

4.5.3 Miscellaneous Migration Handling

Although `ms_routing.c` does not contain primarily socket migration code, it plays an important role in handling socket migrations, from both the migrating end, and from the (usually) stationary end (although the migration system is designed such that it will work even if the processes on both sides of the connection migrate at the same time), as well as the node from which the socket has migrated. The `ms_route_pkt()` function, as well as several lower-level functions it calls, handles several different cases related to socket migration. In the case of the socket opposite the migration, if the socket receives either a valid packet from an address different from the one cached in the socket structure, or a valid packet from the old node with a migration signal in the option field of the packet header, it updates the `rem_cnode` field of the socket as well as the IP address cache. In the case of the node away from which the socket has migrated, as long as the old copy of the socket structure has not yet been deleted, if a packet is received for the socket, the response is a packet with the migration signal in the header. Finally, for the node to which the socket has just migrated, as indicated by the flag `newly_migrated` in the socket structure being set, if a valid acknowledgment is received from the remote socket, that `newly_migrated` flag is cleared and a MIGACKED metasocket message is sent to the node from which the socket had migrated.

The socket's retransmission thread `ms_sock_thread()` in `ms_routing.c` also plays a role in ensuring that socket migrations go smoothly and in preventing data loss when a socket migrates. If the remote socket has migrated while the thread was asleep, any packets that might be on the socket's resend queue are retransmitted immediately, this time to the new address (with a new checksum as appropriate). Also, for a newly-migrated socket, acknowledgment packets are periodically sent to the remote socket until the socket's migration is

acknowledged.

To simplify socket migration, processes are, in certain cases, temporarily prevented from migrating to another node. These are when a process owns a socket that is not in either the Closed or Established state (i.e. when it is in the process of connecting or closing the connection), and when the process owns a connected socket that just migrated but whose migration hasn't yet been acknowledged by the remote socket. The latter case has been mentioned in sections 4.5.1 and 4.5.2; the process is locked from migrating from whichever arrives later—the process, or the socket. The process is then re-allowed to migrate in `ms_routing.c` when a valid acknowledgment is received from the remote socket. All instances of disabling and re-enabling process migration due to socket state are handled as appropriate in `ms_external.c` and `ms_routing.c`.

Chapter 5

Testing and Performance

In this chapter the tests performed to insure the correct functioning of the socket implementation are described. The last of which was also a performance benchmark once the code was working.

5.1 Userspace Unit Tests

At the beginning of this project core portions of the TCP implementation (socket and packet structures, some routing and packet processing code) were written such that they could be compiled in a userspace executable so that unit testing could be performed on at least some of the code. These tests fell into four primary categories: packet cropping, packet signalling, socket hashing, and packet routing.

The user executable `ms_crop_test` and `ms_pcf_crop_test` verified the correct functioning of the `ms_crop_pkt()` function used in handling packets that straddle the edge of either duplicate data or the end of the receive window. Whereas `ms_crop_test` manually created the packet from the test, `ms_pcf_crop_test` added calls to `ms_create_pkt()` and `ms_free_pkt()` to test packet allocation as well.

The programs `ms_signs_test` and `ms_sigrem_test` verified the correctness of the packets created by the functions `ms_signal_nosock()` and `ms_signal()` functions in

`ms_routing.c`, respectively. The first is for replying to an incoming packet that doesn't match the information for a known, and the second is for building packets to reply to an incoming packet addressed to a known socket.

The `ms_hash_test` program verified the proper functioning of the `ms_hash_sock()` and `ms_unhash_sock()` functions in `ms_data.c` for adding sockets to and removing sockets from the system-level socket hash table, as well as proper retrieval from the hash tables with `ms_find_sock_port()` and `ms_find_sock_fd()`.

Finally, `ms_route_test()` created sockets of varying states, added to them to the system hash table and verified the correct response of `ms_route_pkt()` to various valid and invalid incoming packets.

5.2 Kernel Tests

Once the basic socket implementation code was added to the kernel and run successfully, simple client and server programs were written, called `client1` and `server1`. The client would send a text file to the server one line at a time, which would be echoed back by the server and checked by the client to verify that the echoed data matched the original data. At this point, only communication with other sockets on the same node was supported, but this same test program was also used to verify proper communication over the network before adding the actual migration functionality to the implementation.

To test the migration functionality a second version of the client/server test programs was written, called `client2` and `server2`. These programs transmitted the data as blocks instead of lines of text and would migrate periodically as specified on the command line (source code listed in appendix C). For performance comparison against the standard sockets a version of the two programs was written, that did the same thing except using the standard socket system calls. These were called `client2_std` and `server2_std`.

5.3 Performance

As mentioned in section 5.2, `client2/server2` and `client2_std/server2_std` were used to do comparative performance testing on two separate three-node clusters. The first cluster was a set of three eMachines computers with 600 MHz Intel Celeron processors and 160 MB RAM on a 100 megabit network. The second was a set of Atipa dual-processor AMD Athlon machines with 2 GB RAM on a gigabit network (two nodes had 1.8 GHz processors and the other had 2 GHz processors). Files of varying sizes were used for comparison, and in all cases the file would be transmitted three times before migrating to the next node (the migration pattern used is shown in table 5.1), and the time taken for each transmission would be measured on the client side.

	Node 1	**Node 2**	**Node 3**
Run 1	server	client	
Run 2	server / client		
Run 3		server / client	
Run 4		server	client
Run 5			server / client

Table 5.1. Process migration pattern used in performance testing. Nodes 1 and 2 are the home nodes for the server and client processes, respectively.

In the tests run, the time elapsed for each transmission is less a reflection purely of bandwidth but a combination of bandwidth and latency, as the client would wait for the server to echo the last block of data before sending the next block. Nonetheless, to present the performance data in a more "raw", but still readable, form, I have divided the file size by time taken to transmit the file and converted the value into megabits per second.

The table and chart on page 38 show the performance on the cluster of single-Celeron nodes for the standard sockets and of the migratable sockets both initially and after some performance optimizations that are discussed below. As can be expected, there is some

Cur Nodes	Standard (Mb/s)	Mig. Socks before opt. (Mb/s)	imp.	Mig. Socks after opt. (Mb/s)	imp.
1,2	8.25	7.12	−13.73%	7.69	−6.73%
1,1	2.34	3.58	52.84%	3.62	54.53%
2,2	8.22	7.11	−13.51%	7.65	−6.89%
2,3	2.69	3.04	13.08%	3.21	19.44%
3,3	2.74	3.13	14.12%	3.28	19.71%

Table 5.2. Comparative performance on the cluster of single-Celeron nodes, between the standard sockets and migratable sockets, before and after optimizations. (The first column shows which node the server and client are on, respectively, and the "imp." (improvement) columns show the percentage gain or loss compared to using standard sockets.)

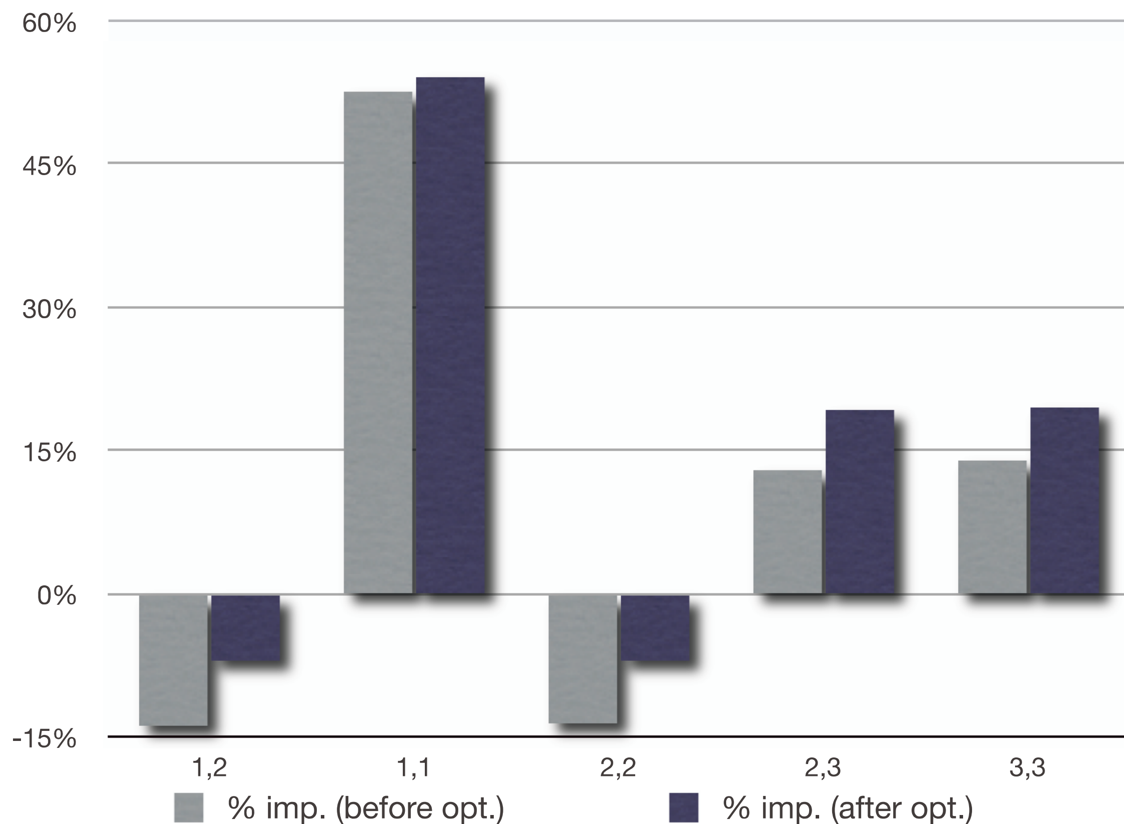

Fig. 5.1. Chart of percentage performance gains and losses on the cluster of single-Celeron nodes before and performance optimizations.

performance loss in comparison to the standard sockets when both processes are on their home nodes. In the test program, the client is stepping through a buffer containing all the data from the file while the server is receiving each block of data into the same part of memory, meaning that more memory is accessed regularly on the client side, which has some interesting performance effects when both processes are moved onto either the server's home node (1,1) or the client's home node (2,2) which could possibly be generalized to the case where one process is primarily sending data (the client in this case) and the other is primarily receiving data (the server in this case). The case where both processes are moved to the client's node appears to be not much different (in terms of performance) from the case where both processes are on their home nodes, but when both processes are on the server's home node performance improves significantly over the standard sockets, even without the performance optimizations. When both processes are moved away from their home nodes (2,3 and 3,3) performance is still improved, but the gain is more modest in comparison to both processes running on the server's home node.

As can be noted from table 5.2 and the chart in figure 5.1, the initial performance was somewhat disappointing, with performance gain when both processes are away from their home nodes being approximately equal to the performance loss when they are both on their home nodes. In an effort to address these issues some effort was put into optimizing for performance. The two greatest inefficiencies in the code at that point were that user data for packets transmitted over the network was being copied twice due to calling a lower-level equivalent of the `sendmsg()` system call that still expected userspace buffers, and that there were a number of places where steps could be short-circuited in the common case, particularly in queuing packets when they did not need to be queued. With these issues addressed, as shown on page 38, the gap in performance lost was closed by just over half, and the performance gain when both processes are moved away from their home nodes (2,3 and 3,3) increased almost by half.

Moving from single-processor computers to dual-processor computers revealed some syn-

Cur Nodes	Standard (Mb/s)	Mig. Socks (Mb/s)	imp.
1,2	34.58	30.71	−11.20%
1,1	6.01	8.42	40.11%
2,2	36.19	32.21	−11.00%
2,3	5.05	5.82	15.29%
3,3	5.01	5.77	15.17%

Table 5.3. Comparative performance on the cluster of dual-Athlon nodes between the standard and migratable sockets. (The first column shows which node the server and client are on, respectively, and the "imp." (improvement) column shows the percentage gain or loss compared to using standard sockets.)

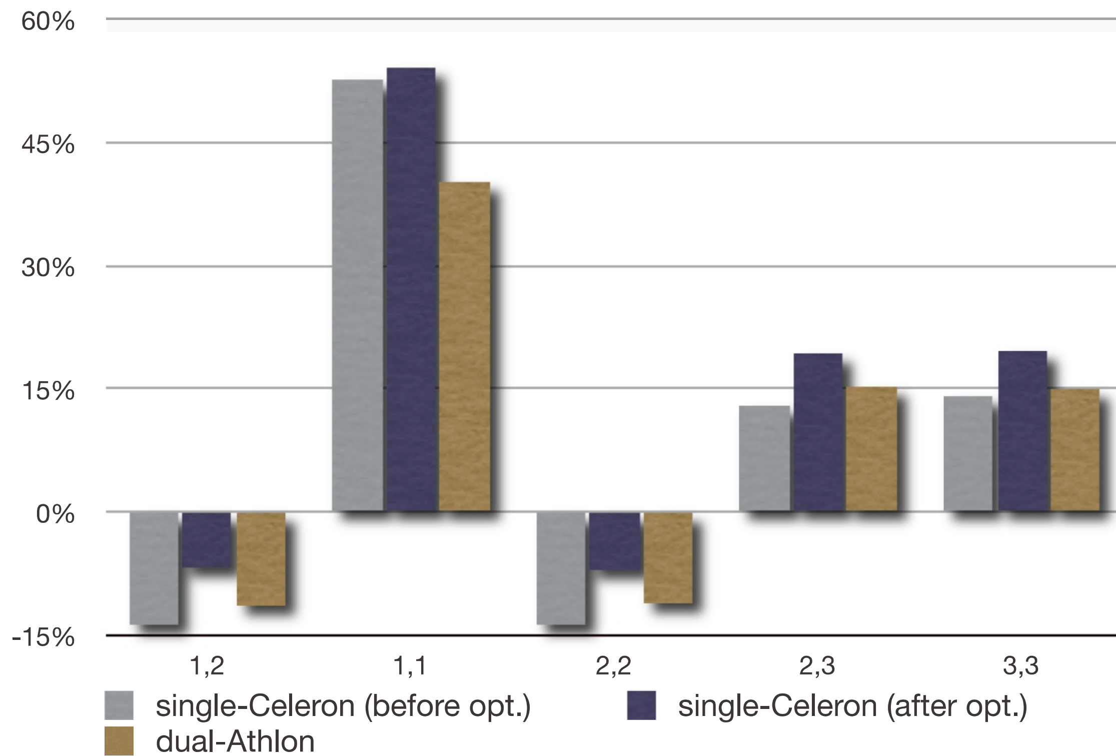

Fig. 5.2. Chart of percentage performance gains and losses on both clusters.

chronization bugs in the code, and after those were corrected additional performance data, shown on page 40, was collected. The performance in comparison to standard socket is significantly lower than on the cluster of single-processor nodes, although still an improvement over the initial relative performance on the eMachines cluster. The drop is most likely due to coarse-grained locking on the migratable sockets compared the existing sockets in the Linux kernel, which has been heavily optimized for multi-processor systems. This is discussed further in the Future Work section.

Chapter 6

Conclusion and Future Work

This chapter gives the conclusions drawn and discusses ways that this project could be improved and expanded.

6.1 Conclusion

Although all of the design goals initially set out for this project were not met, namely full transparency of operation, a working implementation of migrating sockets that do exhibit a performance improvement over the standard nonmigrating sockets was demonstrated. Looking at the design goals laid out in section 3.1 we see that the implementation is, indeed, at the kernel level, and that the socket implementation is fairly self-contained making it relatively portable. As for interoperability, while the sockets implemented are able to only communicate with each other, standard TCP sockets are still available for applications to use for communicating with hosts outside the cluster (the original thought of intercepting the socket system calls was to use the migratable sockets for intracluster communication, but fall back to the standard TCP sockets for communication with hosts outside the cluster). As for maintaining state, the sockets are fully reconstructed on migration (and processes are temporarily prevented from migrating if a socket is neither closed nor fully connected, so as to avoid any odd communication that might result from migrating while the socket

is in an "inbetween" state). Additionally, any outstanding packet queues are reassembled during socket migration, ensuring that no packets are lost on either end of the connection. Because the old copy of the socket is retained on migration until the socket on other end of the connection has acknowledged the migration, each socket is still able find the new location of the other one, allowing for both to migrate simultaneously while maintaining the connection. As for cluster-wide unique addressing, the addition of home nodes to the packet headers allows sockets from differing home nodes to use the same port while temporarily on the same node. Finally, because the current location of a server socket is maintained on its home node, this allows clients to connect even if it has migrated away (because the home node responds with a migration signal).

6.2 Future Work

Port to 2.6 Linux kernel. As I noted in the introduction, in this project I added to the stable version of OpenMosix, which is still tied to version 2.4.26 of the Linux kernel. It would be desirable to move to a more recent version under the 2.6 branch. One possibility here would be to port it to the as-yet unstable version of openMosix, or to port it to another process migration system that works with the 2.6 kernel.

Further performance optimizations. Although some reasonable gains in performance were made, more optimizations can still be made. Particular ways performance could be improved would be by reducing the number of memory allocations (the existing TCP code in the kernel allocates memory for packets by page instead of allocating the packets individually), and, on multiprocessor systems in particular, reducing the socket-access related critical regions as much possible.

System call transparency. As mentioned when discussing the design of this socket implementation, one of the original goals in this project was to allow existing networking applications to take advantage of the socket migration without needing to be rewritten,

and that this is a feature that did not ultimately make it into the project. The sockets, and the handling of their migration, would need to be reworked a bit to achieve this, because the sockets are tied directly to the process using them, which is not true of standard TCP sockets (these are instead associated with the filesystem), making it possible for a server application to hand off a new connection to another process. To accomplish this, the sockets would need to be accessible to any process with same home node as the process that created the socket, and some algorithm for determining which process uses the socket the most and that the socket should therefore follow.

Adapt as back-end for MPI or other message-passing interface. Alternately to the above option, the socket migration facility could be adapted as the back end for an existing parallel software messaging interface such as MPI.

Deeper integration with kernel network stack. If this "MTCP" protocol were adapted into another transport protocol at the same level as UDP and TCP, it would be possible to eliminate the overhead of the UDP header. Another possibility would be to move the home node information into the option field of the TCP header to allow interoperability with standard TCP hosts, where the sockets would fallback into standard TCP/non-migratable functionality when communicating with hosts outside of the cluster.

Bibliography

[1] Kerrighed. http://www.kerrighed.org.

[2] MOSIX: Cluster and Multi-Cluster Grid Management. http://www.mosix.org/.

[3] Open Single System Image Clusters for Linux. http://openssi.org/.

[4] openMosix. http://openmosix.sourceforge.net/.

[5] Andresen, D., Lebak, J., and Bowker, E. An Evaluation of Distributed Scheduling Algorithms Within the DESPOT Architecture. In *the Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)* (June 2004), pp. 186–192. Also available as http://people.cis.ksu.edu/~dan/despot/pdpta04.pdf.

[6] Andresen, D., Schopf, N., Bowker, E., and Bower, T. Distop: A Low-overhead Cluster Monitoring System. In *2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)* (June 2003), pp. 1832–1836. Also available as http://people.cis.ksu.edu/~dan/despot/pdpta03.pdf.

[7] Barak, A., Guday, S., and Wheeler, R. *The MOSIX Distributed Operating System*, vol. 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[8] Barak, A., and La'adan, O. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems 13*, 4-5 (1998). Also available as http://www.mosix.org/pub/mosixhpcc.pdf.

[9] Barak, A., and Litman, A. MOS: A Multicomuter Distributed Operating System. *Software: Practice and Experience 15*, 9 (1985).

[10] Barak, A., and Shiloh, A. The MOSIX2 Management System for Linux Clusters and Multi-Cluster Organizational Grids. http://www.mosix.org/pub/MOSIX2_wp.pdf.

[11] Barak, A., and Wheeler, R. MOSIX: An Integrated Multiprocessor UNIX. In *Proc. Winter 1989 USENIX Conf.* (1989).

[12] Kuntz, B., and Rajan, K. MIGSOCK: Migratable TCP Socket in Linux. Master's thesis, Carnegie Mellon University, 2002.

[13] Postel, J. User Datagram Protocol. ftp://ftp.rfc-editor.org/in-notes/rfc768.txt, Aug. 1980.

[14] Postel, J. Transmission Control Protocol: DARPA Internet Program Protocol Specification. ftp://ftp.rfc-editor.org/in-notes/rfc793.txt, Sept. 1981.

[15] Russell, R. Unreliable Guide to Locking. http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/.

[16] Stevens, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[17] Stevens, W. R. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, 1996.

[18] Stevens, W. R., and Wright, G. R. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

Appendix A

Kernel Programming Interface

The original intention for this project was to make the operation of migratable sockets transparent to user processes by redirecting standard system calls that would operate on any of the "special" sockets to my code, but, as it currently stands, user processes have to use a separate set of system calls to take advantage of socket migration. Many of the functions listed here present a BSD-style socket interface that is currently accessed from a system call (`sys_migsock()`, discussed in section B), but with further work could be called from the standard system calls when the request applies to a migratable socket.

1. `long ms_init(void)`

   Description
   This function initializes the migratable sockets.

   Return Value
   Returns 0 on success or a negative error value on failure.

2. `struct mig_sock *ms_find_sock(unsigned int fd)`

   Input Arguments
   fd The file descriptor of the socket to look for.

   Description
   This function searches for a socket owned by the currently running process that has the specified file descriptor. If it is initially unable to find the socket or the process information, it will sleep up to 500 milliseconds in case the socket is migrating over but hasn't yet completed the migration. If the socket is found, its lock is released before the function returns.

   Return Value
   Returns a pointer to the requested socket if it is found, or a null pointer if the socket is not found.

3. `long ms_listen(unsigned int fd, __u16 lport, struct ms_opts *opts)`

   Input Arguments

fd The desired file descriptor for the new socket.
lport The local port to bind to.
opts A pointer to the socket options to use. Default options are used if it is a null pointer.

Description
Compared to a BSD-style socket interface, this function is roughly equivalent to calling `socket()`, `bind()`, and then `listen()`. It attempts to create a server socket and bind that socket to the requested port. If the currently-running process is not on its home node, the metasocket message `OPENLISTEN` is sent to the home node.

Return Value
Returns 0 on success, or a negative error value on failure. Specific error states handled in the function include existence of a conflicting socket (`EINVAL`), and insufficient memory (`ENOMEM`).

4. `long ms_accept(struct mig_sock *ms, long fd_new, struct sockaddr_in *addr)`

   Input Arguments
   ms A pointer to the listening socket. **fd_new** The desired file descriptor for the new socket if a new connection is established.
   addr A pointer to a user buffer to fill in with the address of the remote host if a new connection is established.

   Description
   This function accepts a connection request on a listening socket. If the socket is not non-blocking, this function will sleep on the socket until there is a new connection request.

   Return Value
   Returns 0 on success, or a negative error value on failure. Specific error conditions handled by this function include the file descriptor in `fd_new` being already in use (`EBADF`), `addr` being an invalid pointer (`EFAULT`), and no pending connections on a non-blocking socket (`EAGAIN`).

5. `long ms_connect(unsigned int fd, __u16 peer, __u16 lport, __u16 rport, struct ms_opts *opts)`

   Input Arguments
   fd The desired file descriptor for the new socket.
   peer The node number of the host to connect to.
   lport The local port to use.
   rport The port to connect to.
   opts A pointer to the socket options to use. Default options will be used if the pointer is null.

Description
This function attempts to create a new socket and connect to the requested host. If this function is called from `sys_migsock()`, the second parameter is taken to be a 32-bit IP address in network byte ordering (big-endian) which `sys_migsock()` converts to a node number.

Return Value
Returns 0 on success, or a negative error value on failure. If the socket is requested to be non-blocking, the function will return as soon as the socket has been created, but before the connection has been established, with the `EINPROGRESS` error value. Specific error cases handled in this function include insufficient memory (`ENOMEM`) and invalid `peer` value (`EINVAL`).

6. `long ms_sendmsg(struct mig_sock *ms, struct msghdr *msg, unsigned int flags)`

   Input Arguments
   ms A pointer to the socket on which to send the data.
   msg A `msghdr` struct pointing to the data to send.
   flags A bitwise OR of flags indicating how the data is to be sent.

   Description
   This function sends data as specified by a `msghdr` structure. The `flags` parameter is roughly equivalent to the one in the `sendmsg()` system call, but the only flag that actually has any effect is `MSG_NOSIGNAL`, which causes a `SIGPIPE` signal not to be sent when the socket is not writable, although it will still return the error value `EPIPE`.

   Return Value
   Returns 0 on success, or a negative error value on failure.

7. `long ms_send(struct mig_sock *ms, void *buf, size_t len, int flags)`

   Input Arguments
   ms A pointer to the socket on which to send data.
   buf A pointer to the data to send.
   len The length of the buffer.
   flags A bitwise OR of flags indicating how the data is to be sent.

   Description
   This function sends data the same way as `ms_sendmsg()`, except that it is passed a pointer to a single buffer to send.

   Return Value
   Returns 0 on success, or a negative error value on failure.

8. `long ms_recvmsg(struct mig_sock *ms, struct msghdr *msg, unsigned int flags)`

Input Arguments
ms A pointer to the socket to read from.
msg A pointer to a `msghdr` structure indicating the buffers into which to copy the data.
flags A bitwise OR of the values in table A.1 indicating how to receive the data.

Description
This function reads data from a socket's receive queue, potentially waiting for data to arrive. If the socket is non-blocking and there is no data in the receive queue, the function will return immediately with an `EAGAIN` error value, unless the `MSG_WAITALL` flag is set.

Return Value
On success, returns the number of bytes read, or 0 if the connection has ended and there is no more data to read. If there is an error, the appropriate negative error value is returned.

Title	**Description**
`MSG_PEEK`	Retrieves data from the receive queue without actually removing that data from the queue.
`MSG_DONTWAIT`	Returns immediately with the `EAGAIN` error value if there is no data in the receive queue.
`MSG_WAITALL`	Waits for enough data to fill the buffer(s).

Table A.1. Flags supported by `ms_recvmsg()` and `ms_recv()`

9. `long ms_recv(struct mig_sock *ms, void *buf, int buflen, unsigned int flags)`

   Input Arguments
   ms A pointer to the socket to read from.
   buf A pointer to the user buffer to copy the data to.
   buflen The length of the buffer in bytes.
   flags A bitwise OR of any of the values in table A.1 indicating how to receive the data.

   Description
   This function works the same way as `ms_recvmsg()`, except that it is passed a single buffer into which to read the data.

   Return Value
   On success, returns the number of bytes read, or 0 if the connection has ended and there is no more data to read. If there is an error, the appropriate negative error value is returned.

10. `long ms_shutdown(struct mig_sock *ms, int how)`

    Input Arguments
    ms A pointer to the socket on which to end the connection.
    how Which part of the connection to end. Valid values are `SHUT_RD`, `SHUT_WR`, and `SHUT_RDWR`, for read, write, and both, respectively.

    Description
    Ends the connection on a socket without destroying the socket itself.

    Return Value
    Returns 0 on success, or a negative error value on failure.

11. `long ms_close(struct mig_sock *ms, int exiting)`

    Input arguments
    ms A pointer to the socket to close.
    exiting A boolean value indicating if the close is due to the process exiting. If the call is from `sys_migsock()` this value is always false and the user is not allowed to pass in this parameter.

    Description
    This function closes a socket.

    Return Value
    Returns 0 on success, or a negative error value on failure.

12. `long ms_exit(void)`

    Description
    Called when the current process is exiting to close all outstanding sockets.

    Return Value
    Returns 0 on success, or a negative error value on failure.

13. `long ms_setsockopt(struct mig_sock *ms, int level, int optname, char *optval, int optlen)`

    Input Arguments
    ms A pointer to the socket to set an option for.
    level The level of the option to set. Recognized values are `SOL_SOCKET`, `SOL_IP`, `SOL_TCP`, and `SOL_FILE`. (`SOL_FILE` is nonstandard and would be accessed by the `fcntl()` system call in a fully-integrated version of this implementation.)
    optname The option to set as listed in table A.2.
    optval A pointer to the new value to set the option to.
    optlen The length in bytes of `optval`.

    Description
    Sets a socket option.

Return Value
Returns 0 on success, or a negative error value on failure.

14. `long ms_getsockopt(struct mig_sock *ms, int level, int optname, char *optval, int *optval)`

    Input Arguments
    ms A pointer to the socket to retrieve an option for.
    level The level of the option to retrieve. Recognized values are `SOL_SOCKET`, `SOL_IP`, `SOL_TCP`, and `SOL_FILE`.
    optname The option to retrieve as listed in table A.2.
    optval A pointer to a buffer into which to copy the option's value.
    optlen A pointer to an `int` variable to set with the option value's length.

    Description
    Retrieve's the value of a socket option.

    Return Value
    Returns 0 on success, or a negative error value on failure.

15. `pid_t ms_get_pid_from_mypid(pid_t mypid, __u16 hnode)`

    Input Values
    mypid The process ID on the process's home node of the process to find.
    hnode The home node of the process to find.

    Description
    Finds the process ID on the current node of a process that has migrated to current node from its home node.

    Return Value
    Returns the process ID of the process in question, or the negative error value `ESRCH` if the requested process is not found.

16. `long ms_process_migrate_inbound(struct task_struct *p, int prev_node)`

    Input Arguments
    p A pointer to the process structure for the process that is migrating onto the current node.
    prev_node The node that the process has migrated away from.

    Description
    Performs the necessary actions and checks for a process that has just migrated onto the current node.

    Return Value
    Returns 0 on success, or a negative error value on failure.

Level	Option	Data Type	Description
SOL_SOCKET	SO_ERROR	int	Read-only. Gets (and resets) the socket's error variable.
SOL_SOCKET	SO_LINGER	struct linger	Gets or sets the linger and timeout settings for closing the socket.
SOL_SOCKET	SO_RCVLOWAT	int	Gets or sets the minimum number of bytes to wait for in a receive call.
SOL_SOCKET	SO_RCVTIMEO	struct timeval	Gets or sets the default timeout in a receive call.
SOL_IP	IP_TTL	int	Read-only. Gets the time-to-live (TTL) value for the underlying UDP socket.
SOL_IP	IP_RECVERR	struct sock_extended_err	Read-only. Gets extended error information on the underlying UDP socket.
SOL_IP	IP_MTU_DISCOVER	int	Read-only. Gets the path-MTU discovery (maximum transmission unit) setting for the underlying UDP socket.
SOL_IP	IP_MTU	int	Read-only. Gets the MTU (maximum transmission unit) of the underlying UDP socket.
SOL_TCP	TCP_LINGER2	int	Number of seconds to wait before forcing a close on an orphaned socket.
SOL_FILE	FILE_FLAGS	int	Gets or sets the file flags for the socket. (The only one observed is O_NONBLOCK.)
SOL_FILE	FILE_SET_FLAGS	int	Write-only. ORs the file flags value with the value passed in.
SOL_FILE	FILE_CLEAR_FLAGS	int	Write-only. ANDs the file flags with the complement of the value passed in.

Table A.2. Socket options supported by `ms_setsockopt()` and `ms_getsockopt()`

17. `long ms_process_migrate_outbound(struct task_struct *p, int new_node)`

    Input Arguments
    p A pointer to the process structure for the process that is migrating away from the current node.
    new_node The node onto which the process is migrating.

    Description
    Performs the necessary actions and checks when a process is migrating away from the current node.

    Return Value
    Returns 0 on success, or a negative error value on failure.

Appendix B

User Application Programming Interface

Access to the socket migration functionality is currently provided through the system call `sys_migsock()` in `ms_systest.c`, which is only compiled if the build option `CONFIG_MOSIX_MIGSOCK_SYSTES` is enabled. Here are the details of `sys_migsock()`:

> `long sys_migsock(int call, unsigned long *args)`
>
> *Input Arguments*
> **call** The function to call, as listed in table B.1. The `MSC_INIT` call is deprecated since `ms_init()` is now called automatically when OpenMosix is initialized.
> **args** A pointer to an array of arguments to pass to the desired function.
>
> *Description*
> The parameters in the `args` array generally map directly to the parameter list of the desired function, with the exception that if the function to be called expects a socket pointer as the first parameter, this function attempts to do a socket lookup, calling `ms_find_sock()`, using the first value passed as the socket's file descriptor.
>
> *Return Value*
> Returns 0 on success or a negative error value on failure.

In my userspace testing code the module `ms_calls.c` contains system call wrappers for each of the functions listed in table B.1. The header and source for this module are listed

Value	Function Called
MSC_INIT	ms_init()
MSC_LISTEN	ms_listen()
MSC_ACCEPT	ms_accept()
MSC_CONNECT	ms_connect()
MSC_SENDMSG	ms_sendmsg()

Value	Function Called
MSC_SEND	ms_send()
MSC_SHUTDOWN	ms_shutdown()
MSC_CLOSE	ms_close()
MSC_SETSOCKOPT	ms_setsockopt()
MSC_GETSOCKOPT	ms_getsockopt()

Table B.1. System call types for `sys_migsock()`.

in sections B.1 and B.2, respectively. Any differences in parameter lists beyond replacing a pointer to the socket to operate on with a file descriptor are mentioned in discussing the appropriate function in section A. If the system call fails, the value returned to the userspace program will be -1, and the global `errno` variable will be set with a positive version of the error value.

B.1 ms_calls.h

```
#ifndef __MSCALLS_H__
#define __MSCALLS_H__

#include <asm/types.h>
#include <netinet/in.h>                                                    5

struct ms_opts {
        int fd_flags;
        int linger;
        int linger_len;                                                   10
        int linger2;
        unsigned int rcvlowat;
        unsigned long rcvtimeo;
        unsigned int sndlowat;
        unsigned long sndtimeo;                                           15
};

long ms_init(void);
long ms_listen(long fd, __u16 lport, struct ms_opts *opts);
long ms_accept(long fd, long fd_new, struct sockaddr_in *addr);           20
long ms_connect(long fd, __u32 ip, __u16 lport, __u16 rport, struct ms_opts *opts);
long ms_sendmsg(long fd, struct msghdr *msg, unsigned int flags);
long ms_send(long fd, void *buf, size_t len, int flags);
```

```
long ms_recvmsg(long fd, struct msghdr *msg, unsigned int flags);
long ms_recv(long fd, void *buf, int buflen, unsigned int flags);
long ms_shutdown(long fd, int how);
long ms_close(long fd);
long ms_setsockopt(long fd, int level, int optname, char *optval, int optlen);
long ms_getsockopt(long fd, int level, int optname, char *optval, int optlen);

#endif
```

B.2 ms_calls.c

```
#include <asm/unistd.h>
#include <hpc/migsock/ms_systest.h>
#include <stdlib.h>
#include <errno.h>
#include "ms_calls.h"

/* The kernel tree pointed to by the makefile MUST HAVE the migsock() syscall
 * configured for this to compile correctly.
 */
static _syscall2(long, migsock, int, call, unsigned long *, args)


long ms_init(void)
{
        return migsock(MSC_INIT, NULL);
}


long ms_listen(long fd, __u16 lport, struct ms_opts *opts)
{
        long args[3];

        args[0] = fd;
        args[1] = (long)lport;
        args[2] = (long)opts;

        return migsock(MSC_LISTEN, args);
}


long ms_accept(long fd, long fd_new, struct sockaddr_in *addr)
{
        long args[3];

        args[0] = fd;
        args[1] = fd_new;
        args[2] = (long)addr;

        return migsock(MSC_ACCEPT, args);
```

```
}                                                                              40


long ms_connect(long fd, __u32 ip, __u16 lport, __u16 rport, struct ms_opts *opts)
{
        long args[5];                                                          45

        args[0] = fd;
        args[1] = (long)ip;
        args[2] = (long)lport;
        args[3] = (long)rport;                                                 50
        args[4] = (long)opts;

        return migsock(MSC_CONNECT, args);
}
                                                                               55

long ms_sendmsg(long fd, struct msghdr *msg, unsigned int flags)
{
        long args[3];
                                                                               60
        args[0] = fd;
        args[1] = (long)msg;
        args[2] = (long)flags;

        return migsock(MSC_SENDMSG, args);                                     65
}


long ms_send(long fd, void *buf, size_t len, int flags)
{                                                                              70
        long args[4];

        args[0] = fd;
        args[1] = (long)buf;
        args[2] = (long)len;                                                   75
        args[3] = (long)flags;

        return migsock(MSC_SEND, args);
}
                                                                               80

long ms_recvmsg(long fd, struct msghdr *msg, unsigned int flags)
{
        long args[3];
                                                                               85
        args[0] = fd;
        args[1] = (long)msg;
        args[2] = (long)flags;

        return migsock(MSC_RECVMSG, args);                                     90
}
```

```
long ms_recv(long fd, void *buf, int buflen, unsigned int flags)
{
	long args[4];

	args[0] = fd;
	args[1] = (long)buf;
	args[2] = (long)buflen;
	args[3] = (long)flags;

	return migsock(MSC_RECV, args);
}


long ms_shutdown(long fd, int how)
{
	long args[2];

	args[0] = fd;
	args[1] = (long)how;

	return migsock(MSC_SHUTDOWN, args);
}


long ms_close(long fd)
{
	return migsock(MSC_CLOSE, &fd);
}


long ms_setsockopt(long fd, int level, int optname, char *optval, int optlen)
{
	long args[5];

	args[0] = fd;
	args[1] = (long)level;
	args[2] = (long)optname;
	args[3] = (long)optval;
	args[4] = (long)optlen;
}


long ms_getsockopt(long fd, int level, int optname, char *optval, int optlen)
{
	long args[5];

	args[0] = fd;
	args[1] = (long)level;
	args[2] = (long)optname;
	args[3] = (long)optval;
	args[4] = (long)optlen;
}
```

Appendix C

Sample Client/Server Application Source

This appendix contains the source code for the final testing and benchmarking program, `client2/server2`. These are contained in four files, `client2.c`, `server2.c`, and the code shared between the two programs, `test_common.c` and `test_common.h`.

C.1 test_common.h

```
#ifndef __TEST_COMMON_H__
#define __TEST_COMMON_H__

void mosctl_init();
unsigned short mosctl_get_home_node();                                                   5
void mosctl_lock();
void mosctl_unlock();
void mosctl_goto(unsigned short node);
unsigned short mosctl_get_cur_node();
#define mosctl_goto_home() (mosctl_goto(0))                                              10
size_t build_msg(int msgtype, unsigned short curnode, void *data, long datalen, void *msgbuf);

typedef struct hdr {
        int msgtype;
        long msglen;                                                                     15
        unsigned short curnode;
} hdr_t;
```

```
#define MSGTYPE_START 2
#define MSGTYPE_DATA 1                                                    20
#define MSGTYPE_END 0

#endif
```

C.2 test_common.c

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>                                                       5
#include "test_common.h"

static char __mosctl_hnode_path[64] = "/proc/hpc/admin/mospe";
static char __mosctl_goto_path[64];
static char __mosctl_where_path[64];                                      10
static char __mosctl_lock_path[64];

static unsigned short __mosctl_hnode;

static void write_num(char *path, int num);                               15
static int read_num(char *path);
static inline void __mosctl_write_num(char *path, int num);
static inline int __mosctl_read_num(char *path);

void mosctl_init()                                                        20
{
        pid_t pid = getpid();
        sprintf(__mosctl_goto_path, "/proc/%d/goto", pid);
        sprintf(__mosctl_where_path, "/proc/%d/where", pid);
        sprintf(__mosctl_lock_path, "/proc/%d/lock", pid);                25
        mosctl_lock();
        __mosctl_hnode = (unsigned short)mosctl_get_home_node();
}

static inline void __mosctl_write_num(char *path, int num)                30
{
        FILE *f = fopen(path, "wt");
        if(NULL==f) {
                printf("Unable to open file \"%s\": %s (errno %d)\n\n", path, strerror(errno), errno);
                fflush(stdout);                                           35
                return;
        }
        fprintf(f, "%d\n", num);
        fclose(f);
}                                                                         40

static inline int __mosctl_read_num(char *path)
```

```
{
        FILE *f = fopen(path, "rt");
        int num = 0;                                                                        45
        if(NULL==f) {
                printf("Unable to open file\"%s\": %s (errno %d)\n\n", path, strerror(errno), errno);
                fflush(stdout);
                return;
        }                                                                                   50
        fscanf(f, "%d", &num);
        fclose(f);
        return num;
}
                                                                                            55
unsigned short mosctl_get_home_node()
{
        return (unsigned short)__mosctl_read_num(__mosctl_hnode_path);
}
                                                                                            60
void mosctl_lock()
{
        __mosctl_write_num(__mosctl_lock_path, 1);
}
                                                                                            65
void mosctl_unlock()
{
        __mosctl_write_num(__mosctl_lock_path, 0);
}
                                                                                            70
void mosctl_goto(unsigned short node)
{
        mosctl_unlock();
        __mosctl_write_num(__mosctl_goto_path, node);
        mosctl_lock();                                                                      75
}

unsigned short mosctl_get_cur_node()
{
        unsigned short node = (unsigned short)__mosctl_read_num(__mosctl_where_path);      80
        return (0==node ? __mosctl_hnode : node);
}


size_t build_msg(int msgtype, unsigned short curnode, void *data, long datalen, void *msgbuf)      85
{
        hdr_t *ptr = msgbuf;
        size_t msglen = (MSGTYPE_DATA==msgtype ? datalen : 0) +
                sizeof(int) + sizeof(long) + sizeof(unsigned short);
                                                                                            90
        ptr->msgtype = msgtype;
        ptr->msglen = msglen;
        ptr->curnode = curnode;
        if(MSGTYPE_DATA==msgtype) {
                ptr++;                                                                      95
                memcpy(ptr, data, datalen);
```

```
        }
        return msglen;
}
```

C.3 client2.c

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>                                                    5
#include <netdb.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>                                                        10
#include <math.h>
#include "ms_calls.h"
#include "test_common.h"

#define PORTNUM 1000                                                       15
#define FD 100
#define BUFSIZE 2000

#define BLOCK_SIZE 1024
                                                                           20
#define FALSE 0
#define TRUE -1

void timeval_subtract(struct timeval *a, struct timeval *b, struct timeval *out);
char *read_file(char *filename, long *len);                                25
int parse_cmdline();
void hex_dump(void *ptr, int len);

int reps_per_node;
unsigned short *nodes;                                                     30
int nodes_len;
__u32 ip;
char *file_to_send;
char *file_data;
long file_len;                                                             35

int main(int argc, char **argv)
{
        int ret;
        int i, j;                                                          40

        printf("Calling ms_init()\n");
        fflush(stdout);
        if(ms_init()<0) {
```

```
		printf("Error %d, %s\n", errno, strerror(errno));                       45
		return 1;
	}

	mosctl_init();
                                                                                        50
	ret = parse_cmdline(argc, argv);
	if(ret!=0) return ret;

	file_data = read_file(file_to_send, &file_len);
                                                                                        55
	/* no options for now */
	printf("Calling ms_connect()\n");
	fflush(stdout);
	if(ms_connect(FD, ip, PORTNUM+1, PORTNUM, NULL)<0) {
		printf("Error %d, %s\n", errno, strerror(errno));                       60
		return 1;
	}

	printf("Entering main loop\n");
	for(i=0;i<nodes_len;i++) {                                                       65
		mosctl_goto(nodes[i]);
		for(j=0;j<reps_per_node;j++) {
			printf("nodes[%d] -- %d, rep %d\n", i, nodes[i], j);
			fflush(stdout);
			if(!send_file()) break;                                          70
		}
		if(j<reps_per_node) break;
	}

	printf("Calling ms_shutdown()\n");                                               75
	fflush(stdout);
	if(ms_shutdown(FD, SHUT_WR)<0) {
		printf("Error %d, %s\n", errno, strerror(errno));
		errno = 0;
	}                                                                                80

	printf("Calling ms_close()\n");
	fflush(stdout);
	if(ms_close(FD)<0) {
		printf("Error %d, %s\n", errno, strerror(errno));                       85
		return 1;
	}

	free(nodes);
	return 0;                                                                        90
}

int send_file()
{
	char msg_buf[BUFSIZE];                                                           95
	void *cur_data = file_data;
	void *msg_data = (void *)((hdr_t *)(msg_buf)+1);
	long cur_ofs = 0;
```

```
long block_size;
long msg_size;                                                                              100
unsigned short local_cur_node, rem_cur_node;
struct timeval start_time, end_time, time_diff;
int ret, col=0, i;
int num_width = (int)ceil(log10((double)file_len)), info_width;
char info_fmt[32];                                                                          105

sprintf(info_fmt, "%%%dd bytes, CN %%d ", num_width);
info_width = num_width + 17;

gettimeofday(&start_time, NULL);                                                            110

while(cur_ofs<file_len) {
        local_cur_node = mosctl_get_cur_node();
        block_size = ((cur_ofs+BLOCK_SIZE)<=file_len ? BLOCK_SIZE : file_len-cur_ofs);
        msg_size = build_msg(MSGTYPE_DATA, local_cur_node, cur_data, block_size, msg_buf); 115
        if(ms_send(FD, msg_buf, msg_size, 0)<0) {
                printf("\n\nSend failed: %s (errno %d)\n\n", strerror(errno), errno);
                fflush(stdout);
                return FALSE;
        }                                                                                   120

        if((ret = ms_recv(FD, msg_buf, BUFSIZE, 0))<0) {
                printf("\n\nReceive failed: %s (errno %d)\n\n", strerror(errno), errno);
                fflush(stdout);
                return FALSE;                                                               125
        }

        if(((hdr_t *)msg_buf)->msgtype!=MSGTYPE_DATA) {
                printf("\n\nServer didn't echo back data!\n\n");
                fflush(stdout);                                                             130
                return FALSE;
        }
        if(msg_size!=((hdr_t *)msg_buf)->msglen) {
                printf("\n\nReceived message size mismatch (sent %d, received %d)\n\n",
                        msg_size, ((hdr_t *)msg_buf)->msglen);                              135
                fflush(stdout);
                return FALSE;
        }
        if(0!=memcmp(cur_data, msg_data, block_size)) {
                printf("\n\nData sent doesn't match data received!\n\nSent data:\n");      140
                hex_dump(cur_data, block_size);
                printf("\nReceived data:\n");
                hex_dump(msg_data, block_size);
                printf("\n");
                return FALSE;                                                               145
        }
        cur_data += block_size;
        cur_ofs += block_size;
        fflush(stdout);
}                                                                                           150

local_cur_node = mosctl_get_cur_node();
```

```
        printf("\ncomplete, CN %02d\n", local_cur_node);
        msg_size = build_msg(MSGTYPE_END, local_cur_node, NULL, 0, msg_buf);
        if(ms_send(FD, msg_buf, msg_size, 0)<0) {                                155
                printf("\nSend failed: %s (errno %d)\n\n", strerror(errno), errno);
                fflush(stdout);
                return FALSE;
        }
        if((ret = ms_recv(FD, msg_buf, BUFSIZE, 0))<0) {                         160
                printf("\nReceive failed: %s (errno %d)\n\n", strerror(errno), errno);
                fflush(stdout);
                return FALSE;
        }
                                                                                 165
        if(((hdr_t *)msg_buf)->msgtype!=MSGTYPE_END) {
                printf("\nDidn't echo end data message!\n\n", strerror(errno), errno);
                fflush(stdout);
                return FALSE;
        }                                                                        170

        gettimeofday(&end_time, NULL);
        timeval_subtract(&end_time, &start_time, &time_diff);
        printf("\n%d sec %d usec\n\n", time_diff.tv_sec, time_diff.tv_usec);
        fflush(stdout);                                                          175

        return TRUE;
}

static void hex_line(void *p, int len, int ofs)                                  180
{
        int i;
        unsigned char *ptr = (unsigned char *)p;

        printf("%04x ", ofs);                                                    185

        for(i=0;i<16;i++) {
                if(i<len)
                        printf("%02x ", ptr[i]);
                else                                                             190
                        printf("   ");
        }
        printf("  ");

        for(i=0;i<len;i++) {                                                     195
                if(ptr[i]>=32 && ptr[i]<=126)
                        printf("%c", ptr[i]);
                else
                        printf(".");
        }                                                                        200
        printf("\n");
}


void hex_dump(void *ptr, int len)                                                205
{
```

```
        int i;

        for(i=0;i<len/16;i++) hex_line(ptr+(i*16), 16, i*16);
        if(len%16 != 0) hex_line(ptr+(i*16), len-(i*16), i*16);                          210
        fflush(stdout);
}

int parse_cmdline(int argc, char **argv)
{                                                                                         215
        struct hostent *hinfo;
        int i, val;

        if(argc<3) {
                printf("Usage:\n  %s {host} {filename} [repetitions per migration] [node 1] [node 2] ...\n\n",
                        argv[0]);
                fflush(stdout);
                return 1;
        }
                                                                                          225
        hinfo = gethostbyname2(argv[1], AF_INET);
        if(h_errno!=0) {
                printf("Unable to resolve %s\n\n", argv[1]);
                fflush(stdout);
                return 1;                                                                 230
        }

        ip = htonl(*(__u32 *)(hinfo->h_addr_list[0]));
        printf("Host resolves to %d.%d.%d.%d, (hex %x)\n", (ip&0xff000000)>>24, (ip&0xff0000)>>16,
               (ip&0xff00)>>8, ip&0xff, ip);                                              235
        fflush(stdout);

        file_to_send = argv[2];

        if(argc>=4) {                                                                     240
                if(sscanf(argv[3], "%d", &reps_per_node)<1 || reps_per_node<1) {
                        printf("Repetitions per migration must be >= 1!\n\n");
                        fflush(stdout);
                        return 1;
                }                                                                         245
        } else {
                reps_per_node = 1;
        }

        nodes_len = (argc>=5 ? argc - 4 : 1);                                             250
        nodes = calloc(nodes_len, sizeof(unsigned short));
        if(NULL==nodes) {
                printf("Unable to allocate memory for nodes list!\n\n");
                fflush(stdout);
                return 1;                                                                 255
        }

        if(argc>=5) {
                for(i=0;i<nodes_len;i++) {
                        if(sscanf(argv[4+i], "%d", &val)<1) {                             260
```

```
                        printf("Usage:\n  %s {host} {filename} [repetitions per migration] [node 1] "
                              "[node 2] . . .\n\n", argv[0]);
                        fflush(stdout);
                        free(nodes);
                        return 1;                                                                    265
                }
                nodes[i] = (unsigned short)val;
        }
    } else {
        *nodes = 0;                                                                                  270
    }
    return 0;
}

void timeval_subtract(struct timeval *a, struct timeval *b, struct timeval *out)                     275
{
    out->tv_sec = a->tv_sec - b->tv_sec;
    out->tv_usec = a->tv_usec - b->tv_usec;
    if(out->tv_usec<0) {
        out->tv_sec--;                                                                               280
        out->tv_usec += 1000000;
    }
}

char *read_file(char *filename, long *len)                                                           285
{
    FILE *in;
    char *buf;

    in = fopen(filename, "r");                                                                       290
    if(NULL==in) {
        printf("Unable to open file \"%s\": %s (errno %d)\n\n", filename, strerror(errno),
              errno);
        fflush(stdout);
        return NULL;                                                                                 295
    }

    fseek(in, 0, SEEK_END);
    *len = ftell(in);
    if(*len<0) {                                                                                     300
        printf("Reading file \"%s\" failed: %s (errno %d)\n\n", filename, strerror(errno),
              errno);
        fflush(stdout);
        fclose(in);
        return NULL;                                                                                 305
    }
    rewind(in);
    buf = malloc((*len)+10); /* pad the length a bit just to be safe */
    if(NULL==buf) {
        printf("Insufficient memory to read file\n\n");                                              310
        fflush(stdout);
        fclose(in);
        return NULL;
    }
```

```
        fread(buf, 1, *len, in);                                    315
        fclose(in);
        return buf;
}
                                                                    320
```

C.4 server2.c

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>                                              5
#include "ms_calls.h"
#include "test_common.h"

#define PORTNUM 1000
#define FD 100                                                       10
#define BUFSIZE 2000

#define FALSE 0
#define TRUE −1
                                                                     15
unsigned short hnode_list[] = { 0 };

int reps_per_node = 1;
unsigned short *nodes = hnode_list;
int nodes_len = 1;                                                   20
int nodes_alloc = FALSE;

int xfer_loop();
int parse_cmdline(int argc, char **argv);
                                                                     25
int main(int argc, char **argv)
{
        long ret, ret2;
        char buf[BUFSIZE];
        struct timeval start_time, end_time, diff_time;              30
        int i, j, data_fin;

        printf("Calling ms_init()\n");
        fflush(stdout);
        if(ms_init()<0) {                                            35
                printf("Error %d, %s\n", errno, strerror(errno));
                return 1;
        }

        mosctl_init();                                               40
```

```
	ret = parse_cmdline(argc, argv);
	if(ret!=0) return ret;

	printf("Calling ms_listen()\n");                                                    45
	fflush(stdout);
	if(ms_listen(FD, PORTNUM, NULL)<0) { /* no options for the moment */
		printf("Error %d, %s\n", errno, strerror(errno));
		return 1;
	}                                                                                   50

	printf("Calling ms_accept()\n");
	fflush(stdout);
	if(ms_accept(FD, FD+1, NULL)<0) {
		printf("Error %d, %s\n", errno, strerror(errno));                           55
		errno = 0;
		printf("Calling ms_close()\n");
		fflush(stdout);
		if(ms_close(FD+1)<0 || ms_close(FD)<0)
			printf("Error %d, %s\n", errno, strerror(errno));                   60
		return 1;
	}

	for(i=0; i<nodes_len; i++) {
		mosctl_goto(nodes[i]);                                                      65
		for(j=0; j<reps_per_node; j++) {
			if(xfer_loop()<0) break;
		}
		if(j<reps_per_node) break;
	}                                                                                   70

	if(i>=nodes_len) {
		while(0==xfer_loop()) { }
	}
                                                                                            75
	fflush(stdout);
	if(ms_shutdown(FD+1, SHUT_RDWR)<0) {
		printf("Error calling ms_shutdown(): %s (errno %d)\n", strerror(errno), errno);
		errno = 0;
	}                                                                                   80

	printf("Calling ms_close()\n");
	fflush(stdout);
	if(ms_close(FD+1)<0 || ms_close(FD)<0) {
		printf("Error: %s (errno %d)\n", strerror(errno), errno);                   85
		return 1;
	}

	return 0;
}                                                                                           90

int parse_cmdline(int argc, char **argv)
{
	struct hostent *hinfo;
```

```
    int i, val;                                                              95

    if(argc>=2) {
        if(sscanf(argv[1], "%d", &reps_per_node)<1 || reps_per_node<1) {
            printf("Usage:\n  %s [repetitions per migration] [node 1] [node 2] ...\n\n",
                argv[0]);                                                    100
            fflush(stdout);
            return 1;
        }
    }
                                                                             105
    if(argc>=3) {
        nodes_len = argc - 2;
        nodes = calloc(nodes_len, sizeof(unsigned short));
        if(NULL==nodes) {
            printf("Unable to allocate memory for nodes list!\n\n");         110
            fflush(stdout);
            return 1;
        }
        nodes_alloc = TRUE;
        for(i=0;i<nodes_len;i++) {                                           115
            if(sscanf(argv[2+i], "%d", &val)<1) {
                printf("Usage:\n  %s [repetitions per migration] [node 1] "
                    "[node 2] ...\n\n", argv[0]);
                fflush(stdout);
                free(nodes);                                                 120
                return 1;
            }
        }
    }
                                                                             125
    return 0;
}

/* returns 0 on successful cycle, returns <0 on error or close request */
int xfer_loop()                                                              130
{
    char msg_buf[BUFSIZE];
    hdr_t *msg_hdr = (hdr_t *)msg_buf;
    int ret, ret2;
                                                                             135
    while((ret=ms_recv(FD+1, msg_buf, BUFSIZE-1, 0))>0) {
        msg_hdr->curnode = mosctl_get_cur_node();

        ret2 = ms_send(FD+1, msg_buf, ret, 0);
        if(ret2<0) {                                                         140
            printf("Error sending reply: %s (errno %d)\n\n", strerror(errno), errno);
            fflush(stdout);
            return -1;
        }
        if(MSGTYPE_END==msg_hdr->msgtype) break;                             145
    }

    return (ret>0 ? 0 : -1);
```

```
}
```